Markt&Technik

Winfried Kassera

TURBO-PASCAL 3.0 auf dem CPC 6128

Ausführlich erläuterte Übungen, Beispiele und Anwendungen für das CP/M Plus-System in über 100 Programmteilen, Prozeduren und Funktionen.

TURBO-PASCAL 3.0 auf dem CPC 6128

Winfried Kassera

TURBO-PASCAL 3.0 auf dem CPC 6128

Ausführlich erläuterte Übungen, Beispiele und Anwendungen für das CP/M Plus-System in über 100 Programmteilen, Prozeduren und Funktionen

Markt & Technik Verlag AG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Kassera, Winfried
Turbo-Pascal 3.0 auf dem CPC 6128 : ausführl. erl. Übungen, Beispiele u. Anwendungen
für d. CP/M Plus-System in über 100 Programmteilen, Prozeduren u. Funktionen / Winfried Kassera. –
Haar bei München : Markt-und-Technik-Verlag, 1987.
ISBN 3-89090-455-6

Die Informationen im vorliegenden Buch werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische
Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Buch gezeigten Modelle und Arbeiten ist nicht zulässig.

Turbo-Pascal® ist ein Warenzeichen von Borland International Inc., USA

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
90 89 88 87

ISBN 3-89090-455-6

© 1987 by Markt&Technik Verlag Aktiengesellschaft,
Hans-Pinsel-Straße 2, D-8013 Haar bei München/West-Germany
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Kösel Kempten
Printed in Germany

Inhaltsverzeichnis

1. Warum Turbo-Pascal? 19

1.1 Vergleich mit BASIC-, Assembler-, Pascal-Stil 19
1.2 Voraussetzungen zum Arbeiten mit diesem Buch 22
1.3 Arbeitsweise, Hinweise, Abgrenzungen 22
1.4 Vereinbarungen, Schreibweisen 23

2. Turbo gut vorbereitet 25

2.1 Turbo-Diskette und -Handbuch 25
2.2 Anlegen einer Turbo-Arbeitsdiskette 26
2.3 Turbo-Testlauf 29
2.4 Turbo marsch! 31
2.5 Der Turbo-Editor 31

3. Turbo-Pascal-Befehle in kleinen Portionen 33

3.1 Programm P1: ClrScr und Write 33
3.2 Programm P2: Konstanten, Write und WriteLn, Stringverknüpfung 38
3.3 Programm P3: Variablen, Read, GotoXY, Zuweisungen 38
3.4 Programm P4: TYPE, BufLen, ReadLn, Stringformatierung 40
3.5 Programm P5: Erfassen von Daten, Felder, Abbruchbedingung, Schleife .. 43

4. Pascal-Prozeduren 47

4.1 Was ist eigentlich eine Prozedur? 47
4.2 Aufbau einer Prozedur 48
4.3 Globale und lokale Variable 48
4.4 Parameter 49
4.5 Programm P6: Prozedur, Set Of, KBD, NOT, OR, AND, Length, ... 49
Parameterübergabe
4.6. Mehrere Parameter, Copy, Val, Length, Formate, Backspace 55
Programm P7: Eingabe von Realzahlen mit PROCEDURE »realin.pd«
4.7 Wichtiges zu den Prozedur-Parametern 60
Programm P8: Variablen- und Festwertparameter

5. Turbo-Prozeduren von Diskette 65

5.1 Der Befehl $I 65
5.2 Verwendung von Include-Dateien 67
5.2.1 Das Rumpfprogramm P9: Koordinaten 67
5.2.2 Umwandlung von Koordinatenform in Dezimalform 70
Prozedur P10 »grad_dez«: Val, Copy
5.2.3 Umwandlung von Dezimalgrad in Grad.Min.Sek. 71
Prozedur P11 »dez_grad«: Int, Frac, Round, Val, Str
5.3 Kompilieren mit den Optionen <C> und <M> 75

6. Funktionen 77

6.1 Was sind Funktionen? 77
6.2 Aufbau einer Pascal-Funktion 78
Funktion P12: Cos, Sqrt, Abs

6.3 Einbau von Funktionen in das Hauptprogramm 81
Programm P13: Streckenberechnung
6.4 Ändern einer Include-Datei während des Programmierens: 83
Die Blockbefehle <CTRL/K>-... des Turbo-Editors
6.5 Zusammenfassung (Funktionen) 85

7. Anwendungsbeispiele für Prozeduren und Funktionen 87

7.1 Die Prozedur »RWKxy.P14« kann einen Kurs berechnen 88
7.2 Die Funktion »kursxy.F15« liefert das gleiche Ergebnis 90

8. Funktion ruft Funktion 95

8.1 Vorsicht – Rekursion! 95
8.2 Quadratwurzel rekursiv berechnet 97
Programm P16: Rekursion, IF..THEN, Ln, Exp
8.3 Anwendung der Standardfunktionen Ln und Exp 103

9. Übersichtliche Daten mit strukturierten Typen 107

9.1 Eindimensionale Felder 108
Programm P17: Array-Demonstration
9.2 Zweidimensionale Arrays (Matrizen) 112
9.3 Array-Konstanten 114
9.4 Programm P18: 2-dim-Array, Arraykonstante, ... 115
geschachtelte FOR-Anweisung
9.5 Frei definierte Datenypen 117
Programm P19: Demo

10. Zauberwürfel, Datenstrukturen und mehrdimensionale Felder 121

10.1 Der Zauberwürfel in verschiedenen Dimensionen 121
10.2 Miniwürfel als eindimensionales Feld 123
Programm P20: selbst definierte Datentypen, Ord
10.3 Versuch mit einem zweidimensionalen Array 125
Programm P21: geschachteltes FOR, Retyping

10.4 Dreidimensionales Array für dreidimensionalen Körper 127
Programme P22 und P23: Array-Konstante, GotoXY, KeyPressed
10.5 Es darf verdreht werden 132
Programm P24/P25: KeyPressed

11. Ein Turbo-Menü 135

11.1 Was darf man denn wählen dürfen? 135
11.2 Eine Menge Tasten, Programm P26: Set Of, IN, IF ... THEN ... ELSE ... 136
11.3 Tastaturabfrage, ... 139
Programm P27: KBD, UpCase, CASE OF, Ord, ASCII-Code
11.4 Aufbau des Menüs, ... 140
Programm P28: WHILE..DO, CASE..OF, UpCase, Exit, Label
11.5 Konzentriert programmieren: 4 Prozeduren in einer 144
P29, P30, P31: Prozedur, Parameter
11.6 Überlassen wir mal etwas dem Zufall 147
Prozedur P32: Randomize, Random(xx), Random, Delete, Copy

12. Zeit ohne Uhr: Arbeiten mit Zeigervariablen 155

12.1 Ein weiterer Datentyp: der Zeiger 156
12.2 Platz für die Zeigervariablen: Heap, New, GetMem 157
12.3 Entfernen unnötiger Variablen 158
12.3.1 Release gibt Heap-Speicherplatz frei 158
12.3.2 Freigeben eines einzelnen Blocks mit Dispose 159
12.3.3 FreeMem schafft Platz 160
12.4 Überprüfungen und Zuweisungen 160
Programm P33: MemAvail, Addr, Ptr
12.5 Zählwerk zur Bestimmung der Zeitdauer 163
Programm P34: Arbeiten mit Zeigern,
Rechnen mit Integerzahlen außerhalb des Standardbereichs

13. Anwendung von Turbo-System-Funktionen 169

13.1 Die ESC-Funktionen von Turbo 170
13.2 Anwendung der ESC-Funktionen – Auflistung 170
13.3 Umschalten des Bildschirmmodus 174
Programm P35.DEM: Modusdemo

13.4 Auswählen des Zeichensatzes 176
Programm P36.DEM: Schriftdemonstration, Umlaute
13.5 Umschalten der Bildschirmfarben, Programm P37.DEM: 177
Hintergrund-, Zeichenfarbe, geschachtelte Prozeduren
13.6 Ein Titelblatt für das Würfelspiel 180
PROCEDURE P38: Farbwechsel, Moduswechsel
13.7 Listing Programm P39: Zauberwürfel (Spiel) 182

14. Erstellung von Bildschirmmasken 189

14.1 Wichtiges über Masken und Fenster 189
14.2 Erstellung einer Eingabemaske 191
Programm P40.DEM: Eingabeschleife, Prozeduraufrufe
14.3 Prozeduren zur Maskenbearbeitung 194
14.3.1 Festlegung der Positionen der Maskenausschnitte 194
Prozedur: »masken_position« (MaskPosi.I41)
14.3.2 Ausdruck der Eingabekommentare 195
Prozedur :»notizen« (Notizen.I42)
14.3.3 Analyse und Bearbeitung der Tastatureingaben 196
Prozedur (Input.I43) »bearbeitung«: Delete, Insert, Ctrl
14.4 Manipulationen mit Stringinhalten 202
14.4.1 Ein Versuch mit FillChar 202
Programm P44: einfache Variable, seltsame Ergebnisse
14.4.2 Eine Anwendung von FillChar 204
Programm P45: FillChar, Zeigertyp, New, Mark, Release, String
14.4.3 Verschiebung eines Speicherbereichs 207
Programm P46: Move, Mem, Zeiger, Midstr, Tausch, Swap

15. Bearbeitung von Dateien 211

15.1 Grundsätzliches über Turbo-Dateien 211
15.2 Eine Datei vom Typ File Of Datentyp 212
Programm P60.DEM: Erfassen, Bearbeiten, Ausgeben
15.2.1 Arbeiten mit Daten eines bestimmten Typs: File Of Datentyp 212
15.2.2 Das Hauptprogramm zu P60.DEM 213
15.2.3 Grobstruktur eines Dateiverwaltungsprogramms 214
Prozedur P47.INC: »FileOfMenu«
15.2.4 Reservieren von Menüzeilen 217
Prozedur P48: DelLine

15.2.5 Erfassen von Daten 218
Prozedur P49.INC: Record
15.2.6 Eröffnung einer Datei 223
Prozedur P50.INC: Assign, Reset, IOresult, $I-, $I+
15.2.7 Speichern einer Datei vom Typ File Of 225
Prozedur P51.INC: Reset, Rewrite, Close, Seek, Record
15.2.8 Laden einer Datei 227
Programmteil P52.INC: Reset, EOF, Close
15.2.9 Ausgabe von Daten auf beliebige Geräte 229
Programmteil P53.INC: Ausgabetreiber
15.2.10 Ein Sortierverfahren: Bubble-Sort 231
Prozedur P54.INC: Hilfsvariable, Copy, Vergleich
15.2.11 Korrigieren eines Datensatzes 236
Prozedur P55.INC: Direktzugriff
15.2.12 Erweitern der Datei 237
Prozedur P56.INC: Flags
15.2.13 Löschen eines einzelnen Datensatzes 239
Prozedur P57.INC: Record-Zuweisungen
15.2.14 Löschen einer ganzen Datei 240
Prozedur P58.INC: Erase
15.2.15 Ein Löschmenü zur Abrundung 240
Prozedur P59.INC: Sub-Menü
15.2.16 Zusammenfassung File Of-Dateien: 242
15.2.17 Listing für Programm P60 (File Of-Datei-Beispiel) 242
15.3 Dateien vom Typ Text 247
15.3.1 Aufbau einer Text-Datei 247
15.3.2 Beispiel für eine Text-Datei 248
15.3.3 Speichern einer Text-Datei 249
Prozedur P61.TXT: WriteLn, Write, Rewrite
15.3.4 Laden einer Text-Datei 251
Prozeduren P62.TXT, P63.TXT: EOF, EOLN, $V-, $V+
15.3.5 Erweitern einer Text-Datei 255
Prozedur P64.TXT
15.4 Dateien vom Typ File (typlose Dateien) 256
15.4.1 Lesen und Schreiben in Blöcken 256
15.4.2 Demo-Programm für den Umgang mit typlosen Dateien 257
Programm P65Block.DEM: Hauptprogramm
15.4.3 Prozedur P66.FIL: KeyPressed 259
15.4.4 Prozeduren P67.FIL und P68.FIL: Puffer füllen und kontrollieren 259
15.4.5 Speichern einer Blockdatei 260
Prozedur P69.FIL: Assign, Rewrite, BlockWrite

15.4.6 Lesen einer Blockdatei von Diskette 262
Prozedur P70.FIL: Assign, Reset, BlockRead

16. Rechnen mit Mengen 265

16.1 Der Mengenbegriff in Turbo 265
16.2 Mengentyp, Mengenvariable, Mengenkonstante 266
16.3 Operationen mit Mengen 268
Beispiel: Lösung von Ungleichungen
16.3.1 Die Mengenverknüpfungen 270
16.3.2 Der Eingabeteil P71.SET: Korrekturglieder 271
16.3.3 Der Lösungsteil: P72.SET 273
16.3.4 Der Lösungsausdruck 275
Prozeduren P73.SET und P74.SET

17. Verknüpfung von Programmen 277

17.1 Einige Hinweise zum Betriebssystem 278
17.2 Der Prozessor des CPC 6128 279
17.3 Die Speicherverwaltung 280
17.4 Untersuchung von Routinen des Betriebssystems 281
17.5 Integrierte Maschinenprogramme 285
Prozedur P77.GRA: Inline, Betriebssystemroutinen
Programm P78.DEM: Grafik-Demo
17.6 Aufruf von CP/M-Routinen 291
Programm P79.DEM: BDOS, Mem, Diskettenspeicherplatz
17.7 Verwendung von BIOS-Funktionen 295
Programm P80.DEM: Lesen eines Sektors, BIOS, BDOS, Register
17.8 Externe Maschinenprogramme 299
Programm P81.DIR: External, BlockRead, Puffer, Dateien
17.8.1 Erstellen von Assembler-Programmteilen, Turbo als Assembler-Editor .. 302
17.8.2 Bereinigen des Objektcodes 306
Programm P83: offene Dateien, File
17.8.3 External-Unterprogramme mit Festwertparametern 308
Parameterübergabe am Beispiel »SEKTOR.EXT«
17.8.4 External-Prozeduren mit Variablenparametern 310
Programm P84 mit External LO85.EXT und LO85.INL
17.8.5 External-Funktionen 313
Programm P86.DEM mit External LO87.EXT

17.8.6 Zusammenfassung und Ergänzungen zu External-Unterprogrammen 314
17.9 External-Anwendungen 316
17.9.1 Bildschirmverwaltung des CPC 6128 316
17.9.2 Speichern einer Bildschirmseite auf Diskette 318
Programm P88.SCR mit External »holblock«
17.9.3 Bildschirmseite von Diskette auf den Schirm 322
Programm P89.SCR mit External-Prozedur »SchreibSchirm«
17.9.4 External-Programme direkt aus dem Arbeitsspeicher 324
17.10 Overlay-Technik spart Speicherplatz 329
17.10.1 Das Prinzip der Overlay-Technik 329
17.10.2 Programmbeispiel P90.DEM: OVERLAY,$I 330
17.10.3 Geteilte OVERLAY-Blöcke, Programm P91.DEM 332
17.11 Verketten von Programmen 334
17.11.1 Execute und Chain 334
17.11.2 Verknüpfung von Programmen mit Execute 335
Programm P92 »EXEPROG«: Assign, File, Execute
17.11.3 Vorbereitung der Execute-Programme 336
17.11.4 Chain-Programme sparen Diskettenplatz 337

Anhänge

A FileHandling – ein nützliches Dienstprogramm 341

B Prozeduren und Funktionen in Turbo-Pascal 347

C Turbo-Compiler-Befehle unter CP/M 80 359

D Tastatur-Codes 361

Das Haupttastenfeld: 361
Numerisches Tastenfeld: 363

E Bedienung des Turbo-Editors 365

Cursorsteuerung 365
Löschen und Einfügen 366
Blockbearbeitung 366
Diverse Kommandos 367

F Literatur **369**

Stichwortverzeichnis **371**

Aufstellung der Programmlistings:

P1 ClrScr und Write 33
P2 Konstanten, Write und WriteLn, Stringverknüpfung 36
P3 Variablen, Read, GotoXY, Zuweisungen 38
P4 TYPE, BufLen, ReadLn, Stringformatierung 40
P5 Erfassen von Daten, Felder, Abbruchbedingung, Schleife 44
P6 Prozedur, Set Of, KBD, NOT, OR, AND, Length, ... 50
P7 Eingabe von Realzahlen mit PROCEDURE »realin.pd« 58
P8 Variablen- und Festwertparameter 61
P9 Koordinaten 67
P10 »grad_dez«: Val, Copy 71
P11 »dez_grad«: Int, Frac, Round, Val, Str 71
P12 Cos, Sqrt, Abs 79
P13 Streckenberechnung 81, 92
P14 Kursberechnung als Prozedur 88
P15 Kursberechnung als Funktion 90
P16 Rekursion, IF..THEN, Ln, Exp 100
P17 Array-Demonstration 110
P18 2-dim-Array, Arraykonstante, ... 115
P19 Demo, frei definierte Datentypen 119
P20 Miniwürfel als eindimensionales Feld 123
selbst definierte Datentypen, Ord
P21 Versuch mit einem zweidimensionalen Array 125
geschachteltes FOR, Retyping
P22 Dreidimensionales Array für dreidimensionalen Körper 128
Array-Konstante, GotoXY, KeyPressed
P23 Fortsetzung von P22 131
P24 Würfel verdrehen, KeyPressed 133
P25 Fortsetzung von P24 133
P26 Tasten, Set Of, IN, IF ... THEN ... ELSE 136
P27 Tastaturabfrage, KBD, UpCase, CASE OF, Ord, ASCII-Code 139

P28 WHILE..DO, CASE..OF, UpCase,Exit, Label 141
P29 Würfel verdrehen rechts – links 144
Prozedur, Parameter
P30 Würfel verdrehen oben – unten 146
P31 Würfel verdrehen vorn – hinten 147
P32 Zufallsfunktionen, Randomize, Random(xx), Random, Delete, Copy 150
P33 Überprüfungen und Zuweisungen, MemAvail, Addr, Ptr 161
P34 Arbeiten mit Zeigern, ... 165
Rechnen mit Integerzahlen außerhalb des Standardbereichs
P35 Umschalten des Bildschirmmodus, Modusdemo 174
P36 Auswählen des Zeichensatzes, Schriftdemonstration, Umlaute 176
P37 Umschalten der Bildschirmfarben 178
Hintergrund-, Zeichenfarbe, geschachtelte Prozeduren
P38 Ein Titelblatt für das Würfelspiel, Farbwechsel, Moduswechsel 180
P39 Zauberwürfel (Spiel) 182
P40 Erstellung einer Eingabemaske, Eingabeschleife, Prozeduraufrufe 192
Programm P40.DEM:
P41 Festlegung der Positionen der Maskenausschnitte 194
Prozedur: »masken_position« (MaskPosi.I41)
P42 Ausdruck der Eingabekommentare 195
Prozedur :"notizen« (Notizen.I42)
P43 Analyse und Bearbeitung der Tastatureingaben 197
Prozedur (Input.I43) »bearbeitung«: Delete, Insert, Ctrl
P44 Ein Versuch mit FillChar, einfache Variablen, seltsame Ergebnisse 203
P45 Eine Anwendung von FillChar, Zeigertyp, New, Mark, Release, String .. 205
P46 Verschiebung eines Speicherbereichs, Move, Mem, Zeiger, Midstr, ... 208
Tausch, Swap
P47 Grobstruktur eines Dateiverwaltungsprogramms, »FileOfMenu« 214
P48 Reservieren von Menüzeilen, DelLine 217
P49 Erfassen von Daten, Record 219
P50 Eröffnung einer Datei, Assign, Reset, IOresult, $I-, $I+ 224
P51 Speichern einer Datei vom Typ File Of 226
Reset, Rewrite, Close, Seek, Record
P52 Laden einer Datei, Reset, EOF, Close 228
P53 Ausgabe von Daten auf beliebige Geräte, Ausgabetreiber 230
P54 Bubble-Sort, Hilfsvariable, Copy, Vergleich 234
P55 Korrigieren eines Datensatzes, Direktzugriff 237
P56 Erweitern der Datei, Flags 238
P57 Löschen eines einzelnen Datensatzes, Record-Zuweisungen 239
P58 Löschen einer ganzen Datei, Erase 240
P59 Ein Löschmenü zur Abrundung, Sub-Menü 241

P60 File Of-Datei-Beispiel 242
P61 Speichern einer Text-Datei, WriteLn, Write, Rewrite 249
P62 Laden einer Text-Datei, EOF, EOLN, $V-, $V+ 251
P63 Einlesen einer Zeile 254
P64 Erweitern einer Text-Datei 255
P65 Demo-Programm für den Umgang mit typlosen Dateien 257
P66 KeyPressed 259
P67 Puffer füllen 259
P68 Puffer auslesen 260
P69 Speichern einer Blockdatei, Assign, Rewrite, BlockWrite 261
Prozedur P69.FIL:
P70 Lesen einer Blockdatei von Diskette, Assign, Reset, BlockRead 262
Prozedur P70.FIL:
P71 Mengenverknüpfungen: Eingabeprozedur 271
P72 Mengenverknüpfungen: Lösungsteil 273
P73 Mengenverknüpfungen: Lösungsausdruck 275
P74 Mengenverknüpfungen: Prozedur »Druck« 276
P76 Darstellung von Maschinencode in Form von Hexzahlen 283
P77 Integrierte Maschinenprogramme, Inline, Betriebssystemroutinen 287
P78 Grafik-Demo 290
P79 Aufruf von CP/M-Routinen, BDOS, Mem, Diskettenspeicherplatz 293
P80 Verwendung von BIOS-Funktionen, ... 297
Lesen eines Sektors, BIOS, BDOS, Register
P81 Externe Maschinenprogramme, External, BlockRead, Puffer, Dateien ... 300
P82 Sektor von Diskette lesen 303
P83 Bereinigen des Objektcodes, offene Dateien, File 307
P84 External-Prozeduren mit Variablenparametern 311
P85 Maschinenprogramm zu P84 312
P86 External-Funktion 313
P87 Maschinenprogramm zu P86 314
P88 Speichern einer Bildschirmseite auf Diskette mit External »holblock« ... 318
P89 Bildschirmseite von Diskette auf den Schirm mit External-Prozedur 322
»SchreibSchirm«
P90 Das Prinzip der Overlay-Technik 330
P91 Geteilte OVERLAY-Blöcke 333
P92 Verknüpfung von Programmen mit Execute 335

A1 Dienstprogramm FileHandling 341

Vorwort

Turbo-Pascal ist ein relativ junger Pascal-Dialekt, der seit 1985 in der Version 3.0 auf dem Markt ist.

Der Aufbau der Programmiersprache Pascal bleibt zwar erhalten, aber die wilhelminische Strenge mit der das »alte« Pascal den Programmierer in die Zucht nahm, ist nun einer milden Überwachung gewichen, so daß nun auch der Anfänger nicht gleich von vornherein abgeschreckt wird, sondern bei längerer Beschäftigung mit dem Turbo-Dialekt zunehmenden Spaß am Programmieren findet, zumal Turbo auch Freiheiten gestattet, die z.B. BASIC nicht bieten kann.

Das Konzept dieses Buches ist daher so angelegt, daß auch der Einsteiger nicht den Mut zu verlieren braucht:

Es fängt mit den einfachsten Turbo-Anweisungen an und erläutert Schritt für Schritt den Aufbau der vorgestellten Programmteile. Außerdem werden mehrere komplette, lauffähige Programme erarbeitet, die so angelegt sind, daß der größte Teil der Turbo-Befehle darin angewendet wird. Die meisten der Unterprogramme sind so aufgebaut, daß sie auch in anderen Programmen sinnvolle Verwendung finden können.

In den letzten Kapiteln dringen wir dann tiefer, aber ebenso verständlich in die nicht ganz so einfachen Turbo-Eigenschaften und -Möglichkeiten ein und lassen uns dabei sogar bis auf die Maschinenebene »hinab«.

Als Handbuchersatz ist dieses Buch zwar nicht gedacht, aber es läßt sich immer dann als Nachschlagewerk verwenden, wenn man Turbo-Probleme lösen muß und dabei als Anschauungsmaterial ein praktisches Beispiel vor Augen braucht.

Eine Einschränkung allerdings haben wir uns auferlegt: Alle Programmbeispiele mit Erläuterungen sind für den Schneider CPC 6128 in der Grundausstattung (ein Laufwerk, monochromer Schirm) ausgelegt, um auch wirklich alle Programme für jedermann lauffähig zu halten. Die Aufnahme der Graphik- und Sound- Befehle hätte den Rahmen dieses Buches gesprengt. Eine Anregung für die Erstellung von Graphikbefehlen bieten wir dennoch. Ansonsten sind wir der Meinung, daß Sie sich bei der Beschäftigung mit diesem Buch so viele Turbo-Kenntnisse aneignen können, daß Sie ohne weiteres mit Turbo-Erweiterungen wie z.B. Turbo-Graphix selbst zurechtkommen werden. Den Befehlssatz dazu finden Sie im entsprechenden Original-Turbo-Handbuch.

Übrigens: Auch Turbo-Pascal lernt man nicht durch Lesen. Selbst das beste Buch kann die Praxis, das Ausprobieren und Variieren vor Ort – nämlich am Computer – nicht ersetzen. Wir wünschen Ihnen dazu viel Vergnügen.

Winfried Kassera

Kapitel

1

Warum Turbo-Pascal?

1.1 Vergleich mit BASIC-, Assembler-, Pascal-Stil

Die meisten Leute, die sich einen Computer anschaffen, wollen damit nicht nur spielen oder mehr oder weniger sinnvolle Tätigkeiten ausüben, sondern sie möchten auch gern die Regie übernehmen und mit Hilfe von selbstgestrickten Programmen ihrer Phantasie oder ihrem Können freien Lauf lassen.

Wer hier tatsächlich Ernst macht, wird sich zwangsläufig einer Sprache bedienen müssen, die der Computer versteht, und das ist in den meisten Fällen BASIC. Es ist relativ einfach zu lernen, hat in den heutigen Versionen genügend Befehle für alle möglichen Bereiche und macht schnell Spaß, weil man sofort Erfolge vorweisen kann.

Leider haben fast alle Dinge im (Computer-)Leben nicht nur Vorteile, sondern meist ebensoviele Nachteile. Bei BASIC ist das die ermüdend langsame Arbeitsweise, wenn es darum geht, längere Rechenoperationen oder Grafikbearbeitungen durchzuführen. Alles läuft zunächst über einen Übersetzer, den sogenannten BASIC-Interpreter, der jeden Befehl erst dem Mikroprozessor und seinen Dienern passend aufbereiten muß.

Abhilfe schafft hier am allerbesten die Programmierung in Maschinensprache mit

Hilfe eines komfortablen Assembler-Programms. Was auf diese Weise programmiert wird, kann an Geschwindigkeit von nichts mehr übertroffen werden. Wem es jetzt noch zu langsam geht, der braucht einen Rechner mit einer höheren Taktfrequenz.

Aber – wie gerade erwähnt – alles hat Nachteile: Assembler ist recht mühsam zu erlernen, man sollte sich dazu im Betriebssystem des verwendeten Geräts auskennen und außerdem die Eigenarten des Prozessors beachten. Setzt man sich nämlich an einen Computer mit einem anderen Mikroprozessor, muß man auch gleich einen neuen Assembler-Code lernen. Beispiel: Der 6510 des C64 benützt eine ganz andere Codierung als der ebenfalls mit 8 Bit arbeitende Z80A des Schneider CPC 6128.

Was tun, um sich elegant aus der Affäre zu ziehen?

Sie ahnen es natürlich schon: Turbo-Pascal lernen. Turbo-Pascal hat den Vorteil, daß man wesentlich ausführlicher und ungezwungener seinen Programmtext schreiben (editieren!) kann, ohne an kleinliche Grenzen zu stoßen. Beispielsweise kann man jede Variable benennen, wie man gerade Lust hat, ohne auf zwei oder drei Zeichen beschränkt zu sein. Maximale Breite heißt eben hier »max_Breite« und nicht nur »mb«, was man nach zwei Wochen nicht mehr entziffern kann. Doch davon mehr bei unseren Beispielprogrammen.

Außerdem zwingt Turbo-Pascal zu einem strengeren Programmierstil, weil der Programmaufbau formal festgelegt ist. Wer von BASIC her gewöhnt ist, im Freistil seine Programme einzutippen, wird sich etwas umstellen müssen. Die Unterprogrammtechnik, die nun verlangt wird, verbietet das spontane Einsetzen von Programmsprüngen. Ein strafferer Aufbau ist aber durchaus empfehlenswert, wenn man den Überblick über ein umfangreiches Programm behalten will.

Mehr Text bedeutet längere Ausführungszeit, denken Sie?

Weit gefehlt. Denn vor das Ausführen wurde das sogenannte Kompilieren gesetzt. Vereinfacht ausgedrückt wird der Pascal-Text in einen prozessorgerechten Maschinencode übersetzt. Dabei ist Turbo-Pascal so geschickt angelegt, daß es gleich alle notwendigen Routinen in einer sogenannten Library (Bibliothek) bereitstellt und auf Wunsch auch dauerhaft beifügt, so daß einem rasanten Übersetzen in den Maschinencode und einem schnellen Programmlauf nichts mehr im Wege steht.

Wer sich mit früheren Pascal-Versionen (UCSD, TCL und wie sie alle heißen) abgeplagt hat, dem wird es ähnlich ergangen sein wie dem Autor: Er hat nach etlichen

Abstürzen und ständigen Mißerfolgen Pascal als wenig effektive Bastelsprache zur Seite gelegt und sich gleich mit Maschinensprache beschäftigt. Als dann Turbo-Pascal auf den Markt kam, war das anfängliche Mißtrauen nicht so schnell zu überwinden. Doch nun kommt man immer mehr auf den Geschmack und kann sich in den meisten Fällen sogar die Assembler-Programmiererei sparen.

Leider gibt es nicht allzu viele methodisch gut aufgebaute Bücher, die sowohl dem Anfänger als auch dem schon etwas Fortgeschrittenen Hilfestellung bieten, um die Stolperstellen zu bewältigen, die jede logische Sprache mit sich bringt und deren es natürlich auch genügend in Turbo-Pascal gibt.

Das Handbuch ist zwar umfassend, aber wenig anschaulich. Die Schulbücher legen Wert auf viel Systematik und sauberen Aufbau, und die Fachzeitschriften liefern Spezialitäten. Wir wollen versuchen, den schwierigen Mittelweg zu finden und für die engagierten Programmierer oder solche, die es werden wollen, das herauszufischen, was wichtig ist.

Damit ist unsere Zielsetzung klar:

Wir wollen nicht das Handbuch ersetzen, sondern die Befehle von Turbo-Pascal mit Beispielen anschaulich belegen. Wir wollen keine Raffinesse um jeden Preis bieten (es ergeben sich ganz von allein solche Situationen), sondern wir wollen soweit wie möglich Einsicht in die Programm-Strukturen vermitteln. Also nicht nur das Wie, sondern vor allem das Warum steht im Vordergrund.

Damit die ganze Geschichte nicht zu langweilig wird, wollen wir solche Programmteile aufbauen, die sich auch zu einem sinnvollen Gesamtprogramm zusammensetzen lassen. Dabei wird sowohl die Arbeit mit Dateien zu ihrem Recht kommen als auch die Mathematik, die ja oft die Voraussetzungen liefern muß, damit irgendwelche Verfahren einwandfrei ablaufen. Doch erschrecken Sie bitte nicht jetzt schon, es wird halb so schlimm.

Vollständigkeit im Umgang mit Turbo oder gar Perfektion streben wir nicht an. Das würde den Rahmen dieses Buches sprengen. Aber wenn Sie die Teilgebiete, die wir ansprechen, durchgearbeitet haben, sind Sie - fast mit Garantie - in der Lage, sich die noch fehlenden Kleinigkeiten mit Hilfe des Handbuches zu erarbeiten.

Konkret gesagt: Wir werden ein Programm zur Berechnung von Strecken und Kursen entwerfen. Damit können Sie aus den geographischen Koordinaten die Entfernungen zwischen beliebigen Orten bestimmen. Und auch die Himmelsrichtung, in der Ihr

Traumziel liegt. Außerdem nehmen wir uns eine nette Simulation vor, die Ihnen bestimmt bekannt vorkommen wird. Mehr davon im weiteren Verlauf.

Einen weiteren Schwerpunkt setzen wir bei der Verknüpfung von Turbo-Text mit Maschinenprogrammen, da dieses Thema meist recht stiefmütterlich behandelt wird. Sie erfahren dort, daß man mit Turbo auch das erledigen kann, was Turbo eigentlich gar nicht kann.

Doch auch wenn Sie diese praktischen Anwendungen nicht oder nur am Rande interessieren, haben Sie nebenbei eine ganze Menge von Turbo und auch von der Erstellung von Lösungsverfahren kennengelernt.

1.2 Voraussetzungen zum Arbeiten mit diesem Buch

- Schneider CPC 6128 mit Laufwerk und monochromem Schirm
- eventuell ein Drucker
- CP/M Plus auf Diskette
- Turbo-Pascal auf Diskette
- Handbuch zu Turbo

Um flotte Fortschritte zu erzielen, sollten Sie schon ein paar Programmierkenntnisse in irgendeiner höheren Sprache aufweisen. Es werden zwar die wichtigen Begriffe erklärt und an Hand von Beispielen erläutert, jedoch sollten Sie z.B. mindestens wissen, was Variablen, Daten, Dateien usw. sind.

Selbstverständlich können Sie auch ohne jegliche Vorkenntnisse hier bei uns mitspielen. Dies wird vielleicht ein etwas mühsamerer, aber dafür schnellerer Wege zum Umgang mit Turbo-Pascal werden.

1.3 Arbeitsweise, Hinweise, Abgrenzungen

Unser Ziel ist es, Ihnen die Möglichkeit zu geben, möglichst viele Begriffe aus Pascal anzuwenden. Dabei gehen wir vor, wie es sich für ein methodisch ordentliches Buch gehört: vom Leichten zum Schwierigen, vom Einfachen zum Komplexen.

Es geht also ganz harmlos los. Aber dabei sollten Sie nicht leichtsinnig werden und sagen: »Das weiß ich ja schon alles.« Vielleicht müssen Sie gerade hier wieder einmal

nachblättern, weil Sie sich irgendeine Schreibweise (Syntax) oder eine Zusammenstellung von Ausdrücken (Semantik) eingebildet haben, die Turbo nicht verträgt. Aber beruhigen Sie sich, uns ist das anfangs genauso ergangen.

In den ersten Beispielen erklären wir sehr gründlich, wenn es sein muß bis zum letzten Strichpunkt, alle Befehle. Später, mit zunehmender Weisheit, halten wir uns ein bißchen zurück und überlassen Ihnen auch ein paar Punkte zum Nachdenken. Hilfestellung bieten wir aber immer. Alle Programmbeispiele sind fortlaufend durchnumeriert und mit einem vorangestellten P oder einem eindeutigen Namen versehen.

Aus Platzgründen wurde – bis auf eine Ausnahme – auf Struktogramme verzichtet. Bei Programmierarbeiten größeren Umfangs ist es aber zweckmäßig, sich solche Ablaufdiagramme zu erstellen, um die Übersicht zu behalten.

Bei der Kommentierung der Programme haben wir auf die Numerierung der Textzeilen und auf den Kommentar an Ort und Stelle zugunsten besserer Programmübersicht verzichtet. Statt dessen finden Sie anschließend an jedes wichtige Programm die notwendigen Erläuterungen für zusammenhängende Sinnabschnitte. Sie werden dieses Verfahren bald angenehm empfinden, weil der Programmtext kompakt bleibt.

Es gibt eine Reihe Unterschiede zwischen Turbo-Pascal und den Standard-Pascal-Dialekten. Wir verschwenden nur selten ein Wort darauf, sondern bemühen uns immer um die Möglichkeiten, die Turbo zur Verfügung stellt.

Wenn irgend möglich, verwenden wir deutschsprachige Ausdrücke und verzichten auf das eingedeutschte Computerkauderwelsch. Jedoch kommen wir nicht darum herum, die notwendigen Fachausdrücke zu verwenden, die sich in vielen Fällen nur schlecht ins Deutsche übersetzen lassen.

1.4 Vereinbarungen, Schreibweisen

Damit wir uns im Text einig sind, vereinbaren wir folgende Schreibweisen:

- Alle Turbo-spezifischen Ausdrücke, die im Begleittext oder im Programmcode vorkommen, sind im Text fett gedruckt. Falls diese Wörter eine Klammerfunktion haben (**BEGIN** und **END** sind z.B. Begrenzer eines Anweisungsteils), sind sie durchgängig in Großbuchstaben geschrieben. Ansonsten ist die Schreibweise die gleiche wie im Handbuch von Turbo-Pascal.

- Selbst definierte oder gewählte Bezeichnungen sind in der Regel in kleinen Buchstaben geschrieben. Im Text sind sie auch nicht fett gedruckt. Das gilt also z.B. für alle Variablennamen oder sonstigen Bezeichner.

- Dateinamen stehen in Großbuchstaben.

- Die ENTER-Taste entspricht der RETURN-Taste.

- Tasten, die zu drücken sind, schreiben wir fett in spitzen Klammern. Beispiel **<RETURN>**

- Gleichzeitig zu drückende Tasten stellen wir in einer gemeinsamen spitzen Klammer dar. Beispiel: **<CONTROL/SHIFT/ESC>**

- Für die Taste CONTROL verwenden wir auch die Abkürzung CTRL.

- Turbo läßt im Programmtext Kommentare zu, die man zwischen geschweifte Klammern setzen kann: `{ Kommentar }`, oder zwischen die runden Klammern mit Stern: `(* Kommentar *)`. Je nach Bedarf verwenden wir beide Arten.

- Im Text werden alle Bezeichner zur Hervorhebung mit den Anführungszeichen »« versehen, während die Beschreibung der Stringinhalte wie im Programmtext mit den einfachen Hochkommas vorgenommen wird.

- In diesem Buch wird – wie auch im Handbuch zu Turbo-Pascal – das Zeichen (^) verwendet. Auf Ihrem CPC entspricht dieses Zeichen der Taste (↑).

- Alle weiteren Besonderheiten besprechen wir an Ort und Stelle.

- Am Schluß des Buches finden Sie die Turbo-Befehle noch einmal in alphabetischer Reihenfolge zusammengestellt und - soweit erforderlich - mit einem kurzen Beispiel erklärt.

Doch genug der Vorreden. Knipsen wir endlich den Hauptschalter für unsere Computeranlage an. Wenn Sie schon mit Turbo gearbeitet haben, können Sie das nächste Kapitel überfliegen. Wir zeigen Ihnen dort nur, wie man sich und das Gerät für die Programmierarbeit präpariert.

Kapitel

2

Turbo gut vorbereitet

2.1 Turbo-Diskette und -Handbuch

Zusammen mit Ihrer Turbo-Diskette haben Sie das Handbuch erhalten, das Ihnen die Turbo-Anweisungen erläutert. Es beschreibt in Kurzfassung alle Möglichkeiten, die Turbo bietet, jedoch ohne jeglichen methodischen Aufbau.

Während dort auf die Abweichungen bei den verschiedenen Betriebssystemen eingegangen wird, beschäftigen wir uns in diesem Buch nur mit der CP/M-80-Version. Das bedeutet, daß Sie sich nicht durch das Handbuch hindurchkämpfen müssen, um durch ständiges Hin- und Herblättern Ihre Turbo-Kenntnisse zu erweitern. Einen methodisch sinnvollen Weg bieten wir Ihnen hier.

Auf der Original-Turbo-Diskette finden Sie eine ganze Reihe von Dateien, von denen wir zunächst nur zwei benötigen: TURBO.COM und TURBO.MSG. Die erstgenannte ist das eigentliche Turbo-Programm mit Editor und Compiler. Die zweite ist eine Datei mit ausführlichen Fehlermeldungen, die beim Compilieren bzw. beim Programmlauf ausgegeben werden. Später – im letzten Kapitel – benötigen wir auch noch TURBO.OVR.

Falls Sie eine Turbo-Pascal-Version haben, die noch nicht auf den Schneider angepaßt ist, dann brauchen Sie die TINST.XXX-Dateien, um für den CPC 6128 die richtigen Bildschirmdaten einzustellen. Dazu mehr im nächsten Abschnitt.

Interessant sind auch die Beispielprogramme mit dem Zusatz .PAS. Wenn Sie etwas mehr Turbo-Erfahrung haben, sollten Sie sich diese einmal anschauen. Besonders das Dienstprogramm LISTER.PAS werden Sie benötigen, weil es Ihnen die Ausgabe eines Listings Ihres Programmtextes auf dem Drucker ermöglicht. Ein einfacher Druckerbefehl dafür existiert nämlich unter Turbo nicht.

2.2 Anlegen einer Turbo-Arbeitsdiskette

Bevor Sie mit Turbo zu arbeiten beginnen, raten wir Ihnen dringend, eine Sicherungskopie von der gesamten Diskette anzufertigen. Das geschieht unter CP/M Plus mit dem Befehl DISCKIT3, nachdem Sie das Betriebssystem mit ICPM aktiviert haben. Die verwendete Diskette ist dabei zunächst im Systemformat zu formatieren, bevor Sie die Taste für den Befehl COPY drücken.

Da es lästig ist, zuerst CP/M Plus zu laden, dann die Turbo-Diskette einzulegen und Turbo zu starten, legen wir uns am besten eine Turbo-Arbeitsdiskette an, die diese Arbeiten von alleine erledigt. Das heißt, wir richten uns eine Diskette her, die nach dem Einschalten des Rechnersystems alle Vorarbeiten von alleine erledigt und Turbo startklar auf den Rechner bringt.

Falls Ihre Version noch nicht angepaßt ist, bringen wir Turbo erst einmal bei, welchen Bildschirm wir besitzen.

Installieren des Bildschirms

Die folgende Arbeit brauchen Sie nur einmal auszuführen:

1. Legen Sie bei aktiviertem CP/M eine Kopie Ihrer Turbo-Diskette in das Laufwerk und tippen Sie **TINST <RETURN>**.

2. Es erscheint ein Menü, bei dem Sie **<S>** für Screen wählen.

3. Nun wird Ihnen ein ganzer Katalog von Bildschirmen zur Auswahl gestellt, von dem Sie die Nummer 20 für den CPC eingeben.

4. Die nächste Frage, ob Sie diese Entscheidung noch einmal ändern wollen, beantworten Sie mit **<N>**, es sei denn, Sie hätten sich vertippt.

5. Bestätigen Sie dann mit **<RETURN>**, daß der Taktzyklus unverändert bleibt.

Wenn das Laufwerk wieder steht, ist das Programm TURBO.COM auf dieser Diskette für den CPC eingerichtet.

Und nun zur Arbeitsdiskette:

Wie man so eine Diskette präpariert, kennen Sie schon, wenn Sie mit CP/M bereits Erfahrungen gesammelt haben. Für diejenigen, die hier noch Schwierigkeiten haben, geben wir vorsichtshalber den Ablauf in Stichworten wieder. Scheuen Sie bitte nicht die Mühe, die Sie nur ein einziges Mal belastet, weil Sie dann von der einmal erstellten Arbeitsdiskette beliebig viele Kopien ziehen können, die Ihnen eine Menge Wartezeit ersparen.

Anweisungen zum Anlegen einer Arbeitsdiskette mit CP/M Plus:

1. Alle Geräte einschalten (am besten über einen einzigen Hauptschalter).

2. CP/M-Plus-Diskette in das Laufwerk einlegen.

3. CP/M mit **|CPM <RETURN>** starten.

4. **DISCKIT3 <RETURN>** holt das Dienstprogramm für die Diskettenbearbeitung in den Speicher.

5. Mit **<4>** Formatieren wählen.

6. Mit **<9>** Herstellung einer CP/M-Diskette fordern und eine beliebige Taste drücken, da bereits eine Diskette mit den notwendigen Systemspuren im Laufwerk liegt.

7. Den weiteren Anweisungen des Programms folgen, die formatierte Diskette entnehmen und mit **<0>** das Programm DISKCKIT3 verlassen.

8. CP/M-Diskette wieder einlegen.

9. **PIP B:C10CPM3.EMS=A:C10CPM3.EMS <RETURN>** kopiert nun die CP/M-Installationen, wenn Sie den Anweisungen zum Wechseln der Disketten richtig folgen.

10. Kopieren Sie auf die gleiche Weise die Datei SUBMIT.COM mit dem PIP-Kommando.

11. Legen Sie die CP/M-Diskette wieder ein, und geben Sie folgendes Kommando ein: **PIP PROFILE.SUB=CON: <RETURN>.** Jetzt geben wir alle Befehle über die Tastatur ein, die beim Start von CP/M automatisch aufgerufen werden sollen. Das ist in unserem Fall zunächst nur **TURBO <RETURN>**.
    Mit **<CONTROL/Z>** steigen wir aus diesem Teil wieder aus.

12. Kopieren der Datei PROFILE.SUB auf die Arbeitsdiskette wie unter Punkt 9 beschrieben.

13. Von der Diskette mit den Turbo-Dateien holen wir uns folgende Teile auf die Arbeitsdiskette:

    ```
    TURBO.COM
    TURBO.MSG
    TURBO.OVR
    ```

    und eventuell noch

    ```
    LISTER.PAS
    ```

    Dazu legen Sie immer zunächst die CP/M-Diskette ein und geben dann z.B. ein:
    PIP A:TURBO.COM=B:TURBO.COM <RETURN>

Wenn Sie sich das Inhaltsverzeichnis mit DIR ansehen und folgende Dateien finden

```
C10CPM3.EMS
SUBMIT.COM
PROFILE.SUB
TURBO.COM
TURBO.MSG
TURBO.OVR
```

dann ist Ihre Arbeitsdiskette bereit.

Dieses Verfahren gestaltet sich wegen des einen Laufwerks etwas umständlich, aber es ist – wie schon erwähnt – nur einmal notwendig. Kopieren Sie nun diese Diskettenseite auf all die Disketten, mit denen Sie Turbo-Programme zu erstellen beabsichtigen.

2.3 Turbo-Testlauf

Wenn Sie nun Ihr Gerät aus- und wieder einschalten oder besser noch einen RESET mit den Tasten **<CONTROL/SHIFT/ESC>** durchführen, dann brauchen Sie ab sofort nur noch eine Arbeitsdiskette einzulegen und mit **|CPM** zu starten. Nach den üblichen Einschaltmeldungen meldet sich Turbo mit der Frage auf dem Schirm:

```
Include error messages (Y/N)?
```

Beantworten Sie das mit **<Y>**, dann wird die Fehlerdatei geladen, die Ihnen beim Programmieren sicher nützlich ist.

Die nun folgende Menütafel ist im Handbuch genau beschrieben. Wir beschränken uns daher auf eine Kurzfassung und suchen uns nur die Punkte heraus, die uns schnell zur Arbeit mit Turbo führen:

<L> Laufwerknummer anmelden (wir bleiben zunächst auf Laufwerk A)

<W> Arbeitsdatei benennen (max. 8 Zeichen, der Zusatz PAS wird automatisch vorgenommen)

Beispiel: LISTER

<M> Hauptdatei benennen, unter der die Arbeitsdatei behandelt werden soll (z.Z. auch nicht erforderlich)

<E> Sprung in den Turbo-Editor

Wenn Sie vorher LISTER gewählt haben, erscheint der Turbo-Text für dieses Programm. Wenn nicht, werden Sie vorher noch einmal nach der Arbeitsdatei gefragt.

Aus dem Editor kommen Sie wieder mit der Tastenfolge **<CTRL/K>** und **<D>** heraus. Mit einem weiteren Tastendruck sind Sie wieder im Menü.

<C> Start des Compilers

Starten Sie ruhig einmal die aktuelle Arbeitsdatei LISTER. Sie sehen die Zeilennummern, die gerade übersetzt werden, auf dem Schirm. Nach wenigen Sekunden ist LISTER startbereit.

<R> Wenn Sie mit **<R>** das Programm LISTER.PAS starten, dann werden Sie nach der zu druckenden Datei gefragt.

Geben Sie hier wieder LISTER.PAS ein, denn eine andere steht uns im Moment noch nicht zur Verfügung. Auf den Zusatz .PAS darf diesmal nicht verzichtet werden.

Wenn Sie Ihren Drucker angeschlossen und eingeschaltet haben, dann erhalten Sie nun einen Ausdruck des Listings des Turbo- Programms LISTER.PAS. Sie bekommen damit gleich einen Eindruck dessen, was Sie mit Turbo erwartet. Falls Ihnen das alles etwas sonderbar vorkommt, lassen Sie sich bitte nicht von ungewohntem Text erschrecken. Schließlich haben wir ja mit Turbo eigentlich noch gar nicht angefangen.

<S> Sie speichern mit dieser Taste grundsätzlich die Arbeitsdatei unter ihrem aktuellen Namen auf dem Laufwerk, unter dem sie angemeldet war.

<D> Ausgabe des Inhaltsverzeichnisses mit eventueller Maske.

Drücken Sie **<RETURN>**, dann erscheint das komplette Hauptverzeichnis.

Sie können aber mit der Option »dir mask« auch das Laufwerk wählen, indem Sie z.B. »B:« eintippen. Wollen Sie die CP/M-Dateien unterdrücken, geben Sie z.B. *.PAS ein, und Sie erhalten nur die Dateien mit dem Zusatz .PAS angezeigt.

<O> wählt die Art des Kompilierens mit einem kleinen Sub-Menü:

<M> kompiliert und beläßt das Kompilat im Speicher.

<C> legt ein lauffähiges Kompilat auf Diskette unter dem Namen der Arbeitsdatei, aber mit dem Zusatz .COM ab.

<H> legt ein Rumpf-Kompilat auf Diskette ab.

Wir belassen es zunächst bei der Voreinstellung auf **<M>**.

<Q> Verlassen von Turbo, zurück zu CP/M.

Wenn Sie wieder mit Turbo programmieren wollen, dann schreiben Sie hinter das Bereit-Zeichen einfach Turbo und schließen mit **<RETURN>** ab.

<X> startet ein Programm mit dem Zusatz .COM als Maschinenprogramm und kehrt dann zu Turbo zurück. Dazu muß auf der Arbeitsdiskette allerdings die Datei TURBO.OVR vorhanden sein.

2.4 Turbo marsch!

Der vorausgegangene Abschnitt hatte zum Ziel, Ihnen den Start mit Turbo zu erleichtern. Um die Turbo-Programme zu schreiben, die wir Ihnen in den nächsten Kapiteln vorstellen werden, brauchen Sie jetzt nur folgendes zu tun:

- Geräte einschalten
- Turbo-Arbeitsdiskette einlegen
- **|CPM <RETURN>** eingeben
- Fehlerdatei mit **<Y>** anfordern
- mit **<E>** den Turbo-Editor anfordern
- einen Namen für das zu bearbeitende Turbo-Programm angeben

Anmerkung: Mit einer angeforderten Fehlerbehandlung haben Sie 28778 Byte Platz im Arbeitsspeicher, ohne Fehlerdatei dagegen 30224 Byte, die Sie aber wahrscheinlich nicht brauchen werden.

2.5 Der Turbo-Editor

Der eingebaute Editor, den Sie vom Menü aus mit **<E>** erreichen, ist nichts anderes als ein Textprogramm, mit dem Sie über den Bildschirm den Arbeitsspeicher füllen.

Sie schreiben wie in BASIC gewohnt Ihre Programmzeilen (ohne Zeilennummern) und schließen sie mit der Rücklauftaste **<RETURN>** ab. Den Cursor bewegen Sie dabei ähnlich wie in WordStar mit Hilfe der Taste **<CTRL>**, die Sie mit dem linken kleinen Finger gedrückt halten, während Sie die gewünschte weitere Taste tippen. Oder aber Sie verwenden die Cursortasten unterhalb des numerischen Tastenblocks.

Falls Ihnen dies fremd ist, werden Sie ein bißchen Übung brauchen, bis Sie die einzelnen Tasten im Griff haben. Sie finden dazu eine kurze Übersicht im Anhang.

Ebenfalls über **<CTRL>** können Sie einzelne Zeichen oder ganze Textblöcke einfügen, löschen, versetzen, speichern, kopieren usw. Auch dazu schauen Sie bitte im Anhang nach. Wenn Sie ausführlichere Anweisungen benötigen, sollten Sie sich das entsprechende Kapitel aus dem Handbuch zu Gemüte führen.

Auf den nächsten Seiten erhalten Sie immer wieder einzelne Hinweise, so daß Sie sich jetzt unbeschwert an Turbo heranwagen sollten, auch wenn Ihnen noch nicht alles geläufig ist.

Kapitel

3

Turbo-Pascal-Befehle in kleinen Portionen

3.1 Programm P1: ClrScr und Write

Ein erstes Miniprogramm soll uns den Einstieg in die Turbo-Welt erleichtern. Sie haben Ihr Turbo betriebsbereit geladen und wählen aus dem Turbo-Menü das **<W>** für Workfile, also die Arbeitsdatei. Als Filenamen schlagen wir »P1« vor.

Tippen Sie nun folgenden Text ein. Schließen Sie jede Zeile mit einem **<RETURN>** ab:

```
(*A*) PROGRAM P1; (* ClrScr, Write *)
(*B*) BEGIN
(*C*)   ClrScr;
(*D*)   Write (^J^J^J);
(*E*)   Write ('Dieser Satz wird auf dem Schirm ausgegeben.')
(*F*) END.
```

Erläuterungen:

(A) Unser Programm beginnt mit der Kopfzeile **PROGRAM P1;**, wobei das Wort **PROGRAM** (Achtung, nur mit einem »M«) bereits zur Syntax von Turbo gehört. Der Zusatz (hier »P1«) ist frei wählbar. Diese Zeile kann auch entfallen.

(B) Der eigentliche Start findet im Anweisungsteil statt, der mit dem Befehl **BEGIN** beginnt.

(C) **ClrScr** bedeutet Bildschirm löschen (clear sreen).

(D) **Write** entspricht dem BASIC-Befehl »PRINT«, also Ausgabe auf dem Schirm, wenn kein anderes Gerät aktiviert ist.

Die Zeichen ^J stehen für Zeilenvorschub.

(E) Der nächste **Write**-Befehl gibt den Text aus, der zwischen den Hochkommas (Akzentzeichen) steht.

(F) Jedes Programm endet mit dem Wort **END** und einem Punkt dahinter.

Zur Form:

- Jede Anweisung muß mit einem Strichpunkt abgeschlossen werden, sonst bricht Turbo die Übersetzung an dieser Stelle ab und weist Sie konsequent darauf hin, daß hier ein »;« erwartet wird.

- Der **Write**-Befehl akzeptiert nur das, was zwischen runden Klammern steht. Klartexte müssen dabei zwischen einfachen Hochkommata stehen.

- Sparen Sie nicht am Platz durch enges, unübersichtliches Schreiben Ihres Pascal-Textes. Gönnen Sie jeder Anweisung eine eigene Zeile, es sei denn, zwei oder mehr dieser Befehle hängen eng zusammen.

- Rücken Sie nach jedem **BEGIN** zwei Zeichen ein, bis Sie wieder auf das Schlußwort **END** stoßen. Das steht dann wieder genau unter dem **BEGIN**.

- Kommentare können Sie jederzeit an beliebigen Stellen des Programmtextes unterbringen. Wir schreiben sie zwischen Klammern mit Stern oder in geschweifte Klammern. Solche Zeilen werden nicht durch einen Strichpunkt abgeschlossen.

Starten des Programms:

Drücken Sie nun die Tasten **<CTRL/K>** und **<D>**, dann steigen Sie aus dem Editor aus, d.h., das Schreiben des Programmtextes ist hiermit zunächst einmal beendet.

Die Taste **<R>** (für RUN) bewirkt nun das Kompilieren des Pascal-Textes. Falls irgend etwas nicht stimmen sollte, wird abgebrochen, und Sie werden auf einen Fehler hingewiesen.

Im Falle einer Fehlermeldung tippen Sie auf **<ESC>**, und schon steht der Cursor wieder im Programmtext hinter der Stelle, die nicht einwandfrei war. Sie müssen also Ihren Fehltritt immer vor dem Cursor suchen! Tun Sie das, indem Sie die fehlerhaften Zeilen einfach überschreiben oder ergänzen. **<RETURN>** ist nicht notwendig, um eine Zeile abzuschließen. Es kann sogar zu unerwünschten Effekten kommen, weil mit **<RETURN>** die Zeile dort beendet wird, wo der Cursor gerade steht. Der Rest rechts davon wird in eine neue, eingeschobene Zeile gesetzt.

Die Verbesserungen brechen Sie mit der Tastenfolge **<CTRL/K>** und **<D>** ab. Kompilieren Sie dann zur Abwechlsung das Ganze einmal nur mit **<C>**.

Der Unterschied zwischen einem Start mit **<C>** und **<R>** liegt darin, daß zwar in beiden Fällen eine Umsetzung in ein lauffähiges Programm erfolgt, daß es aber mit **<R>** sofort danach auch gestartet wird, falls die Fehlerüberprüfung keinen Abbruch erzwungen hat.

Mit unserem kleinen Programm P1 ist natürlich noch nicht viel Staat zu machen: Der Schirm wird gelöscht, und der vorprogrammierte Text erscheint. Dieses Ergebnis dürfte zwar niemanden vom Hocker reißen, aber es war zumindest einmal ein Anfang. Der übliche, wenn man mit einer neuen Sprache beginnt.

Wenn Sie wollen, dann speichern Sie diese erste Routine ab, indem Sie hinter dem Prompt-Zeichen »>« die Taste **<S>** drücken.

Andernfalls wählen wir **<E>** und gelangen wieder in den Turbo-Editor, der den bisherigen Programmtext nicht vergessen hat. Wollen Sie ihn nicht mehr sehen, dann brechen Sie mit **<CTRL/K>** und **<D>** ab, wählen mit **<W>** eine neue Arbeitsdatei, und schon geht das Spiel von vorne los.

Für alle weiteren Beispiele gilt ab sofort das gleiche, so daß wir uns fortan diese Hinweise sparen können.

3.2 Programm P2: Konstanten, Write und WriteLn, Stringverknüpfung

Im nächsten Programmbeispiel kommen wir zu einer Pascal-typischen Eigenart: Alle Ausdrücke, die später verwendet werden sollen, müssen vor dem eigentlichen Anweisungsteil, also vor dem ersten **BEGIN**, festgelegt (deklariert, erklärt, definiert) werden, falls es nicht schon Wörter sind, die Turbo kennt.

Beginnen wir mit der Deklarierung von feststehenden Ausdrücken, also solchen, die sich während des Programmlaufs normalerweise nicht verändern.

Hier unser Turbo-Beispiel:

```
      PROGRAM P2;
(*A*) CONST
        satz1='Das ist der erste Satz. ';
        satz2='Und nun der zweite. ';
(*B*)   satz3:String[50]='Ende. ';
      BEGIN
        ClrScr;
(*C*)   WriteLn('(*C*):');
        WriteLn(satz1);
        WriteLn(satz2);
        Write(satz1);
        Write(satz2);
        WriteLn(satz3);
(*D*)   WriteLn('(*D*):');
        WriteLn(satz1,satz2,satz3);
(*E*)   WriteLn('(*E*):');
        Write(satz1+satz2+satz3);
(*F*)   satz3:='Fortsetzung folgt.';
        WriteLn(satz3);
(*G*)   WriteLn('(*F*)  Verknuepfung mit Concat:');
        satz3:=Concat(satz1,satz2,satz3);
        Write(satz3);
      END.
```

Erläuterungen:

(A) Dem Anweisungsteil geht ein sogenannter Deklarierungsteil voraus, in welchem wir zunächst die konstanten Strings »satz1« und »satz2« definieren. Dieser Teil wird mit **CONST** eingeleitet. Diese Konstanten besitzen nun einen Wert, der nicht mehr durch eine neue Zuweisung verändert werden kann. Das heißt, eine

Anweisung wie z.B. satz1:='Neuer Satz eins.' ist nicht möglich und führt zu einer Fehlermeldung.

(B) Ebenfalls unter **CONST** ist es aber auch möglich, sogenannte typisierte Konstanten zu deklarieren und ihnen dabei gleich einen Anfangswert zuzuweisen (Initialisierung). Diese Art kann behandelt werden wie Variablen. Man kann also »satz3« aus unserem Beispiel während des Programmlaufs neu belegen (siehe Punkt F).

Den typisierten Konstanten muß wiederum ein bestimmter Typ zugewiesen werden. In unserem Falle wollen wir eine Zeichenkette erklären, also wählen wir dafür das Pascal-Wort **String**. In eckigen Klammern geben wir an, wie lang dieser String maximal sein kann.

Dabei spielt es keine Rolle, ob wir einen Typ angeben, der länger ist als notwendig. Probieren Sie aber mal aus, was passiert, wenn er kürzer ist!

Hinter jeder Konstantenerklärung erwartet der Compiler einen Strichpunkt.

(C) Der Ausdruck auf dem Schirm erfolgt nun auf verschiedene Weisen. Sie sollten dabei auf jeden Fall folgendes feststellen:

Die Kommentarklammern und -sterne haben innerhalb eines Strings keine Wirkung, sondern werden wie alle anderen Zeichen bearbeitet. Das gilt im übrigen auch für alle reservierten Turbo-Wörter wie z.B. **PROGRAM**, **CONST** usw.

Die Anweisung **Write** beläßt den Cursor hinter dem letzten ausgegebenen Zeichen des Textes, während **WriteLn** (von »Write Line") immer an den Anfang der nächsten Zeile springt.

(D) Hinter **Write** oder **WriteLn** können beliebig viele Ausdrücke stehen. Sie müssen – bis auf einige Ausnahmen – durch Kommata getrennt werden.

(E) Strings lassen sich genau wie in BASIC mit dem Operator »+« verknüpfen. Mehrere Zeichenketten werden dabei zu einer einzigen zusammengefaßt.

(F) Zur Demonstration, daß die typisierte Konstante »satz3« eigentlich eine Variable ist, belegen wir sie mit einem neuen Inhalt, den wir anschließend ausgeben.

(G) Eine weitere Stringverknüpfung bietet Turbo mit **Concat**. »satz3« wird hier wie eine Variable durch eine sogenannte Zuweisung belegt und enthält nun die Zeichen von »satz1«, »satz2« und »satz3« in dieser Reihenfolge, sofern die Länge von »satz3« dies erlaubt.

Zur Form:

- Lassen Sie nach jedem Programmteil eine Zeile frei. Das erhöht wesentlich die Lesbarkeit und hebt die Programmstruktur hervor.
- Beginnen Sie auch nach dem einleitenden Wort **CONST** für jede zu deklarierende Konstante mit einer neuen, eingerückten Zeile.
- Typisierte Konstanten werden verwendet, um Speicherplatz zu sparen, wenn man in einem Programm sehr oft die gleichen festen Werte benötigt.
- Gewöhnliche Konstanten sind entweder Strings oder Zahlen, während typisierte Konstanten alle Typarten zulassen.
- Turbo verträgt beliebig viele Leerzeichen zwischen den Anweisungen, aber auch zwischen **Write** oder **WriteLn** und dem zugehörigen Parameter.
- Keine Leerzeichen dagegen dürfen innerhalb eines Bezeichners wie z.B. »satz2« stehen.

3.3 Programm P3: Variablen, Read, GotoXY, Zuweisungen

Im nächsten Schritt werden wir unseren Text frei gestalten können. Dazu brauchen wir einen Befehl, der dem »INPUT« aus BASIC entspricht. Das ist die **Read**-Anweisung. Probieren wir mal folgendes aus:

```
      PROGRAM P3; (* Variablen, Read, GotoXY, Zuweisung *)
(*A*) VAR
        satz:String[30];
      BEGIN
        ClrScr;
```

```
(*B*)   Write('Geben Sie hier einen Text ein: ');
        Read(satz);
(*C*)   satz:='*****   '+satz+ '   *****';
        ClrScr;
(*D*)   GotoXY(20,13);
        Write(satz)
      END.
```

Erläuterungen:

(A) Der erste Teil des Programms besteht diesmal aus einem VAR-Deklarierungsteil, in welchem die Variable »satz« für Turbo verständlich eingerichtet wird. Sie ist als String-Typ mit einer maximalen Länge von 30 Zeichen definiert.

(B) Der **Read**-Befehl entspricht in etwa dem »INPUT« aus BASIC. Er fällt unter die Gruppe der Prozeduren, mit denen man normalerweise Dateien bearbeiten kann.

Wenn nichts anderes vereinbart wurde (wie, das werden wir später noch sehen), dann nimmt unser System eine Standarddatei an und nennt sie **Input**. Jetzt wird von der Tastatur eingelesen. Damit der Benutzer auch weiß, was er da alles eingetippt hat, kommt als sogenanntes Echo die Bildschirmausgabe zurück. Also wird auch gleichzeitig eine Standard-Ausgabedatei geführt, eben der Schirminhalt, die sich dann **Output** nennt.

(C) Die Variable »satz« kann beliebig verändert werden. Das geschieht durch die sogenannte Zuweisung eines Wertes, wobei hier unter Wert nicht nur ein Zahlenwert gemeint ist, sondern eben alles, was die Variable aufnehmen kann.

Zuweisungen erfolgen immer durch einen Doppelpunkt mit nachfolgendem Gleichheitszeichen (:=). Man spricht dies auch als »...wird belegt mit....« aus.

Bitte beachten Sie, daß in unserem Fall durch die Anbringung der Verzierungssterne 14 Zeichen zusätzlich zur ursprünglich eingetippten Länge unseres Satzes hinzukommen. Was meinen Sie, was passiert, wenn Ihr Satz schon 25 Zeichen lang war? Jawohl, nur die ersten 30 von den nun 39 Zeichen werden weiter verwaltet, den Rest vergißt Turbo. Das darf es auch, denn wir haben ihm ja ausdrücklich genehmigt, daß es nur 30 Zeichen für die Variable »satz« zu berücksichtigen braucht.

(D) Mit dem **GotoXY**-Befehl kann man jede Bildschirmstelle erreichen, indem man in runden Klammern zuerst die gewünschte Spalte und danach die Zeile angibt. Der Cursor wird in unserem Beispiel auf Spalte 20 in der Zeile 13 gesetzt.

Zur Form:

- Für Turbo spielt es keine Rolle, ob Sie Ihren Programmtext mit kleinen oder großen Buchstaben schreiben. Solange Sie die Zeichen nicht in Strings einbauen, werden Groß- und Kleinbuchstaben gleichwertig behandelt.

- Zweckmäßigerweise wählen Sie eine Ihnen angenehme, leichtlesbare Form und schreiben am besten mit kleinen Buchstaben, heben aber mit den großen Wichtiges hervor, z.B. die Turbo-Wörter und -Bezeichner.

- Bei den Zuweisungen verlangt Turbo, daß Doppelpunkt und Gleichheitszeichen unmittelbar hintereinander stehen. Beide Zeichen gehören also streng zusammen und werden in dieser Form nur bei Wertzuweisungen verwendet.

3.4 Programm P4: TYPE, BufLen, ReadLn, Stringformatierung

Im Versuchsprogramm P3 können Sie bei der Eingabe mit **Read** auch mehr als 30 Zeichen eintippen. Bei der Ausgabe werden aber immer nur die ersten 30 Stellen berücksichtigt. Probieren Sie das mal aus!

Gegen überlange Eingaben sichern wir uns ab, indem wir die Länge des Eingabepuffers begrenzen, über den beim **Read**-Befehl die Zeichen von der Tastatur hereingeholt werden. Erst mit Drücken der Abschlußtaste wird nämlich bei der Standardeingabe der Variablen ein Wert zugewiesen, vorher wird alles erst einmal gesammelt, weil der Eingeber an der Tastatur seine Meinung durch Korrigieren des Textes ja noch ändern könnte.

```
      PROGRAM P4;
(*A*) TYPE
        lang=String[20];
        kurz=String[10];
      VAR
        satz:lang;
        wort:kurz;
```

```
        BEGIN
          ClrScr;
(*B*)     BufLen:=20;
          Write('Satz eingeben: ');Read(satz);
(*C*)     BufLen:=5;
          Write('Wort eingeben: ');ReadLn(wort);
          Write('Ausdruck erfolgt unten!');
          GotoXY(1,20);
(*D*)     Write(satz:40);
          Write(wort:10);
        END.
```

Erläuterungen:

(A) Diesmal wird im Deklarierungsteil zunächst der Datentyp vordefiniert, den wir bei der Variablendefinition brauchen. Dieses Vordeklarieren hat den Vorteil, daß man Variablen vom gleichen Typ mit einer kurzen Typerklärung versehen in einer Zeile zusammenfassen kann.

Übrigens spielt hier Turbo seine große Stärke aus. Wir sind nämlich nicht nur auf die wenigen vordefinierten Datentypen beschränkt, sondern können uns (fast) völlig frei beliebige Arten von Daten ausdenken.

Ein Vergleich mit BASIC lohnt sich. Dort gibt es nur die Typen String, Real und Integer. Turbo dagegen hat schon mal eine ganze Reihe Standardtypen mehr zu bieten: **Integer**, **Real**, **String**, **Boolean**, **Char**, **Byte** und **Text**. Und außerdem die Möglichkeit, selber welche zu schaffen. Wir kommen später noch darauf zurück.

(B) **BufLen** begrenzt die Länge des Eingabepuffers auf einen zugewiesenen Wert. Deswegen auch die Zeichen »:=«.

Diese Begrenzung gilt aber nur für das nachfolgende **Read** oder **ReadLn**. Wird nichts angegeben, dann sind maximal 127 Zeichen möglich.

(C) Beim Programmlauf erkennen Sie unschwer den Unterschied zwischen **Read** und **ReadLn**: Letzteres setzt nach der Eingabeprozedur den Cursor immer an den Beginn der nächsten Zeile. Das geschieht also analog zu den Ausgabebefehlen **Write** bzw. **WriteLn**.

(D) Die Ausgabe eines Strings (und nicht nur die) läßt sich formatieren: Setzt man hinter den auszugebenden String oder die entsprechende Variable einen Doppelpunkt mit einer Zahl, so werden bei der Ausgabe so viele Stellen berücksichtigt, wie die Zahl angibt. Sollte sie größer sein als die Länge der ursprünglichen Zeichenkette, dann werden links die restlichen Stellen mit Leerzeichen aufgefüllt (rechtsbündige Ausgabe in diesem Zeichenfeld). Hat der String jedoch mehr Stellen als das Format vorsieht, dann beginnt die Ausgabe mit dem ersten Zeichen des Strings (linksbündige Ausgabe).

Das hat aber nichts damit zu tun, wie die Länge des Strings definiert wurde. Sie sehen bei der Ausgabe von »wort«, daß dort zehn Zeichen erlaubt sind, obwohl der Typ von »wort« nur ein String der Länge 5 ist. Die Geschichte mit dem Doppelpunkt und der Zahl bezieht sich ausschließlich auf die Ausgabe.

Wenn Sie aber mit **BufLen** eine längere Eingabe erlauben, als mit dem Typ vereinbart wurde (z.B. BufLen:=15 bei Read(wort), wenn »wort« vom Typ String[10] ist), dann merken Sie erst bei der Ausgabe, daß »kurz« nur 10 Zeichen aufnehmen kann. Beim Eingeben reklamiert Turbo nicht.

Zur Form:

- Die Definition der Datentypen mit **TYPE** muß immer vor dem Anweisungsteil, also vor dem ersten **BEGIN** durchgeführt werden, wenn die Datentypen in diesem Block benützt werden sollen.

- Das gleiche gilt für die Deklarierung der Variablen mit **VAR** und der Konstanten mit **CONST**. Allerdings verzichtet Turbo auf eine feste Reihenfolge dieser drei Deklarierungen. Zweckmäßigerweise beginnt man jedoch mit **TYPE**.

- Der Datentyp einer Variablen läßt sich entweder bereits unter **TYPE** festlegen, wie wir das in Programm P4 getan haben, oder die Variable wird erst im Teil **VAR** mit ihrem Typ versehen.

- Turbo verlangt bei der Typerklärung das Gleichheitszeichen (=), bei der Definition von Variablen aber einen Doppelpunkt.

- Mehrere Variablen vom gleichen Typ werden durch Kommata voneinander getrennt, bevor Doppelpunkt und Typbezeichner folgen.

Beispiel: VAR satz1,satz2,satz3: lang; oder VAR satz1,satz2,satz3: String[50];

3.5 Programm P5: Erfassen von Daten, Felder, Abbruchbedingung, Schleife

Bis jetzt haben wir nur sehr kleine Schritte unternommen, um in Turbo-Pascal voranzukommen. Jetzt wagen wir mal etwas mehr: Unser Ziel ist es, Daten über die Tastatur zu erfassen. Und zwar wollen wir den Namen eines Ortes eingeben und danach seine geographische Lage in Längen- und Breitengraden. Zum Beispiel wäre Paris (Flughafen Charles de Gaulle) ein markanter Punkt in Europa. Er liegt auf 49 Grad nördlicher Breite und auf 2 Grad 32 Minuten östlicher Länge.

Unsere Eingabe sähe dann etwa so aus: »Paris 49.00 Nord, 02.32 Ost«.

Bevor wir uns das Programm anschauen, vorab noch ein paar Bemerkungen zum Begriff des Feldes oder **Array**. Wir werden diesem wichtigen Thema ein eigenes Kapitel widmen, aber zunächst sollen Sie einmal ein bißchen schnuppern. Erschrecken Sie bitte nicht, wenn Ihnen das alles ein wenig spanisch vorkommt. Es ist alles halb so schlimm, wie es aussieht. Sie müssen nur eins tun: unseren Gedankengängen konsequent folgen.

Unter einem Feld versteht man zunächst einen Satz – also eine feste Anzahl – Daten, die alle vom gleichen Typ sind. Diese Daten werden in abzählbarer Form geordnet, so daß man also jederzeit sagen kann: »Gib mir mal das zehnte Dingsda (Datum) aus der Kiste (Feld) heraus.«

Nehmen wir es tatsächlich heraus und behalten seine Nummer 10 bei, dann wird es auch genau wieder an derselben Stelle abgelegt, falls wir es nicht mehr benötigen. Wir können vorher das Aussehen von Element Nummer 10 verunstalten wie wir wollen, solange es die Nummer 10 auf dem Rücken trägt, behält es auch eisern seinen Platz im Feld.

Die Ortsnamen, die wir gleich eingeben werden, numerieren wir auf diese Weise und legen dafür ein Feld an. Für die zugehörigen Erdnetzkoordinaten »Breite« und »Länge« tun wir das gleiche. Damit erhalten wir drei Felder, nämlich **Array Of ort**, **Array Of breite** und **Array Of laenge**.

Logischerweise werden dem ersten Ort auch die jeweils ersten Koordinaten zugeordnet, dem zweiten Ort die zweiten usw. Das muß zwar nicht so sein, das ist aber genau der Sinn der Felder. Das schafft Ordnung und Übersicht. Wer andere aber heillos verwirren will, kann sich ohne weiteres eine andere Ordnung ausdenken. Mit solchen Vertauschungen läßt sich eine relativ knacksichere Verschlüsselung von Daten erreichen.

Unter einer Feldvariablen jedenfalls werden die einzelnen Komponenten des Feldes zusammengefaßt. Das Auseinanderhalten geschieht mit Hilfe der ordnenden Zahl (Ordnungszahl), die in eckigen Klammern hinter dem Feldvariablennamen steht. Diese Zahl kann natürlich wiederum von einer Variablen vertreten werden. Nehmen wir an, Paris ist der zehnte Ort, den wir in unsere Sammlung aufgenommen haben, dann sieht das auf Pascal so aus:

```
ort[10]='Paris'; breite[10]='42.00.00'; laenge[10]='02.32.00'
```

Alle Pariser Daten laufen hier unter der Nummer 10. Das ist praktisch.

Doch nun endlich zu unserem Pascal-Text:

```
      PROGRAM P5; (* Array, REPEAT..UNTIL, IF..THEN *)
(*A*) TYPE
        tex=String[15];
        grad=String[8];
(*B*) VAR
        ort:    Array [1..100] Of tex;
        breite: Array [1..100] Of grad;
        laenge: Array [1..100] Of grad;
(*C*)   n: Integer;
      BEGIN
        ClrScr;
(*D*)   n:=0;
(*E*)   REPEAT
(*F*)     n:=n+1;
          Write(n,'. Ort (Abbruch mit x): ');
          BufLen:=15;
          Read(ort[n]);
(*G*)     IF ort[n] <> 'x' THEN
          BEGIN
            Write('    Nord: ');
            BufLen:=8;
            Read(breite[n]);            (* Austauschzeilen *)
            Write('    Ost: ');
            BufLen:=8;
            Read(laenge[n]);            (* Austauschzeilen *)
            WriteLn;
          END;
        UNTIL ort[n]= 'x';
(*H*)   DelLine;
        WriteLn(^J'Eingabe beendet.');
      END.
```

Erläuterungen:

(A) Wir definieren zunächst zwei Datentypen: einen für die Bezeichnung des Ortes und einen für die Eingabe der Koordinaten.

(B) Als Variablen wählen wir die eben angesprochenen Felder. Die Angabe, von welcher Nummer ab gezählt werden soll, und die letzte benützbare Nummer sind obligatorisch in der Form [minimum..maximum] aufzuführen. Wir haben also die Eingabe von insgesamt genau hundert Ortsangaben zugelassen.

(C) Die Variable »n« nehmen wir als Zähler her, um unsere Feldelemente zu ordnen. Typ ist das standardisierte **Integer**, also die Ganzzahlen.

(D) Nach dem ersten **BEGIN** setzen wir zunächst den Zähler auf Null, denn er wird zu Beginn der folgenden Schleife sofort auf 1 erhöht.

(E) Mit dem Wort **REPEAT** (Wiederhole) leiten wir die eigentliche Schleife ein, die so lange abgearbeitet wird, bis die Bedingung erfüllt ist, die nach **UNTIL** (bis..) steht. Als Abbruchbedingung haben wir hier die Eingabe von »x« als Ortsnamen gewählt.

(F) Damit beim nächsten Durchlauf auch das nächste Feldelement behandelt wird, muß der Zähler »n« jedesmal erhöht werden.

(G) Damit nur dann weitergearbeitet wird, wenn die Abbruchbedingung nicht erfüllt ist, setzen wir **IF...THEN** ein. Das entspricht dem gleichen Befehl wie in BASIC.

Aber in Pascal haben wir die Möglichkeit, daß wir nun zwischen einem **BEGIN** und einem **END** mit Strichpunkt beliebig viele Anweisungen ablaufen lassen können, ohne durch irgendwelche Zeilenlängen beschränkt zu sein.

Solange also kein Abbruch gewünscht wurde, werden auch die Koordinaten des eingegebenen Ortes verlangt. Ein Ort mit Namen »x« scheidet nun allerdings aus. Aber welcher heißt schon so?

(H) Im Falle des Abbruchs löschen wir die nicht benötigte Eingabezeile mit **DelLine** und geben die Meldung aus, daß die Eingabetätigkeit, also das Erfassen der Daten, beendet ist.

Zur Form:

- In allen Programmteilen, die Unterprogrammcharakter haben, rücken wir um mindestens zwei Zeichen im Text ein, und zwar so lange, bis wir auf das Endwort des eben durchlaufenen Anweisungsblocks stoßen. Das erhöht ganz wesentlich die Übersicht.

- Bei richtiger Wahl der Einrücklängen sind wir mit dem letzten **END** wieder ganz am linken Rand angelangt.

- Einen Abschnitt haben wir mit Kommentarzeilen als »Austauschzeilen« markiert. Hier sind Verbesserungen möglich und notwendig, die wir in den folgenden Abschnitten vornehmen werden.

- Vielleicht haben Sie sich gewundert, daß hinter dem **END** in der **IF**-Anweisung ein Strichpunkt steht. Das muß so sein; denn wenn Sie einen Punkt setzen, erkennt der Compiler dies als Ende des gesamten Programms und hört hier auf zu arbeiten.

Sie sollten nun dieses Programm P5 ausprobieren und vor allem mit den bisher erläuterten Befehlen variieren.

Kapitel

4

Pascal-Prozeduren

4.1 Was ist eigentlich eine Prozedur?

Eine Prozedur läßt sich ohne weiteres mit einem Unterprogramm vergleichen. Aber statt mit einem GOTO- oder einem GOSUB-Befehl, wie beispielsweise in BASIC, ruft man eine Prozedur in Pascal mit ihrem Namen auf, den man frei wählen kann.

Turbo-Pascal arbeitet dann brav alle Anweisungen der aufgerufenen Prozedur ab, kehrt – wie beim BASIC-Befehl RETURN – wieder in das ursprüngliche Programm zurück und holt sich die nächste Anweisung, die unmittelbar hinter dem Prozeduraufruf folgt.

Selbstverständlich können Prozeduren auch geschachtelt werden. Das bedeutet lediglich, daß eine weitere Prozedur von einer Prozedur aus aktiviert wird. Die Rückkehr erfolgt dann entsprechend.

Prozeduren sind also Pascal-Unterprogramme, wenn auch reinrassige »Pascalianer« den Begriff Unterprogramm verabscheuen.

4.2 Aufbau einer Prozedur

Der Aufbau einer Prozedur wird Ihnen im folgenden Beispiel sehr rasch bekannt vorkommen: Er entspricht nämlich ziemlich genau dem bisher besprochenen Programmaufbau. Lediglich steht am Anfang das reservierte Wort **PROCEDURE**, hinter dem ein frei gewählter Name angegeben werden muß. Danach folgen in hoffentlich geübter Weise der Deklarierungs- und der Anweisungsteil.

Wir haben schon einmal darüber gesprochen, daß sogar Turbo-Pascal nicht alles mit sich machen läßt, wenn es um Ordnung und Disziplin geht. Hier bei den Prozeduren finden wir dafür ein prägnantes Beispiel: Jede Prozedur muß erstellt (deklariert) werden, bevor man sie aufrufen kann. Und das ist wörtlich und örtlich gemeint. Sie muß demnach im Programmtext vor dem Aufruf stehen.

Gewöhnen wir uns also schnellstens das von BASIC geförderte unübersichtliche Schreiben von Unterprogrammen irgendwo im Text ab und überlegen uns vorher, was wir eigentlich programmieren wollen.

4.3 Globale und lokale Variable

Wenn wir nachher das Beispielprogramm P6 besprechen, werden Sie sich vielleicht wundern, warum sowohl im Hauptprogramm als auch in der Prozedur Variablen eingerichtet werden. Das hat seinen guten Grund:

Alle Variablen, die vor den Prozeduren deklariert werden, gelten für das gesamte Pascal-Programm, von der ersten **BEGIN**- bis zur letzten **END**-Anweisung. Überall, wo sie auftreten, können sie mit Werten belegt oder mit anderen Konstanten und Variablen verknüpft werden und nehmen dann einen eventuellen neuen Wert an. Diese Art von Variablen nennt man global.

Werden aber nach einem Prozedurnamen Variablen eingerichtet, dann gelten diese nur für diese Prozedur. Solche sogenannten lokalen Variablen sind außerhalb der Prozedur unbekannt. Das geht so weit, daß Pascal sogar gleiche Namen (Bezeichner) für verschiedene Variablen im Hauptprogramm und in den Prozeduren zuläßt. Man kann innerhalb einer Prozedur noch soviel mit einer Variablen »XY« anstellen, der Inhalt einer gleichnamigen globalen Variablen bleibt davon unberührt.

Ist jedoch nur einmal »XY« erklärt worden – und zwar als globale Variable –, dann wird sie auch in den Prozeduren erkannt.

4.4 Parameter

Prozeduren können sogenannte Parameter verarbeiten. Das ist außergewöhnlich praktisch. Man übergibt vom Hauptprogramm aus der aufgerufenen Prozedur einen Wert, den aktuellen Parameter, der dann in der Prozedur verwendet wird. Meist ist dies eine Variable, die be- oder verarbeitet werden soll. Es können aber auch mehrere sein. Wichtig ist nur, daß die Reihenfolge der Parameter beim Aufruf genau derjenigen im Kopf der Prozedur entspricht.

Weil dies ein wichtiges Pascal-Thema ist, werden wir uns im nächsten Abschnitt eingehender damit befassen. Zunächst genügt es zu wissen, daß eine Prozedur einen Wert aus dem aufrufenden Programmteil übernehmen und mit ihm operieren kann.

Doch nun zu unserem Beispiel. Wenn Sie es gewissenhaft durcharbeiten, und wenn Sie ein wenig experimentierfreudig sind, haben Sie in kurzer Zeit Spaß an den Prozeduren, die noch weitere erfreuliche Eigenschaften aufweisen. Aber eins nach dem anderen.

4.5 Programm P6: Prozedur, Set Of, KBD, NOT, OR, AND, Length, Parameterübergabe

Erinnern Sie sich an Programm P5? Dort wurden unter anderem die Koordinaten eines Ortes eingelesen. Normalerweise sollte man diese Eingabe in der Form »Grad.Minuten.Sekunden« vornehmen, also z.B. für Paris Nord 49.00.00 und Ost 02.32.00. Um später mit diesen Koordinaten rechnen zu können, ist diese Form wünschenswert. Bei solchen ungewohnten Eingaben kann aber jeder leicht ein paar Tippfehler machen, die sehr unangenehm werden können, wenn sie weiterverarbeitet werden.

Doch wozu haben wir unseren Computer? Wir lassen Fehler dieser Art erst gar nicht zu:

- Für die Gradzahl geben wir z.B. zwei Stellen frei, so daß bis 99 Grad getippt werden kann.

- Für die Minuten und die Sekunden dürfen wir dagegen nur Eingaben bis jeweils 59 zulassen, da 60 Minuten gleich ein Grad und 60 Sekunden gleich eine Minute sind.

- Außerdem verlassen wir uns nicht darauf, daß der Programmanwender die benötigten Trennpunkte schreibt, sondern wir setzen sie dann, wenn sie an der Reihe sind.

Die Eingabe soll sich auch durch Verwenden bestimmter Tasten wieder korrigieren lassen, wenn mal eine Ziffer danebenging, und erst wenn die **RETURN**-Taste gedrückt wurde, wird die Koordinateneingabe abgeschlossen und in einer Variablen abgelegt.

Das Ganze sieht dann so aus:

```
        PROGRAM P6;

(* 1.Teil: globale Deklarierungen *)

        TYPE
          grad=String[8];
        VAR
          breite: grad;

(* 2.Teil: Deklarierung der Prozedur »gradin« *)

        PROCEDURE gradin (VAR grin:grad);
        VAR (* Deklarierung der lokalen Variablen *)
          z: Integer;
          ziffern: Set Of Char;
          ch: Char;
(*A*)   BEGIN
          ziffern:=['0'..'9'];
          grin:='';
          z:=0;
(*B*)     REPEAT
(*C*)       Read(KBD,ch);
(*D*)       IF ch IN ziffern THEN
            BEGIN
(*E*)         IF NOT (((z=2) OR (z=4)) AND (ch > '5')) THEN
              BEGIN
                z:=z+1;
                Write(ch);
                grin:=grin+ch;
(*F*)           IF (z=2) OR (z=4) THEN
               BEGIN
                 Write('.');
                 grin:=grin+'.';
               END;
             END;
           END;
(*G*)     UNTIL (ch=^M) OR (Length(grin)=8);
        END; (* Ende der Prozedur »GRADIN« *)
```

```
(* 3.Teil: eigentliches Hauptprogramm *)

        BEGIN
          ClrScr;
          Write('Breite: '); gradin(breite);
          ClrScr; GotoXY(20,12);
          Write('eingegebene Koordinate: ',breite);
        END.
```

Erläuterungen:

- Wie üblich werden zunächst die Datentypen festgelegt und die Variablen erklärt. Im ersten Teil wird also die globale Variable »breite« definiert.

- Eigentlich geht es jetzt im 3.Teil mit dem Hauptprogramm los: »gradin« ruft nun die gleichnamige Prozedur auf und übergibt die Variable »breite« an die Variable, die im Kopf der Prozedur hinter dem Prozedurnamen angegeben ist. In unserem Fall handelt es sich um »grin«. »breite« und »grin« sind vom gleichen Typ.

- Die Prozedur »gradin« benötigt die Variablen »z«, »ziffern« und »ch«. Das sind die lokalen Variablen, die nur innerhalb der Prozedur aktiv sind.

- Neu ist hier der Typ **Set Of** und **Char**.

  Mit **Set Of** lassen sich Mengen von irgendwelchen Gebilden beschreiben (zu Mengen siehe Kapitel 16). Wir benötigen hier die sogenannten Characters, also die Zeichen, die unser Rechner ständig verwaltet (u.a. die ASCII-Zeichen).

- »ch« ist die Variable, die ein einzelnes dieser Zeichen erfassen kann.

Erläuterungen zu den einzelnen Programmteilen:

(A) Aus der Menge aller druckbaren Zeichen wählen wir die aus, die für die Eingabe von Koordinaten notwendig sind, also alle Ziffern.

Wir bilden also eine Teilmenge der Characters. Da diese durch den Code geordnet sind, brauchen wir sie nicht einmal alle einzeln aufzuführen. Es genügt, wenn wir den Bereich durch die Grenzelemente abstecken und genau zwei Punkte dazwischensetzen.

(B) **REPEAT** leitet die Hauptschleife zum Einlesen der Zeichen ein. Zuvor wird der Ziffernzähler »z« auf Null gesetzt und dieVariable »grin« entleert. Turbo-Pascal vergißt nämlich die Variableninhalte auch bei einem erneuten Start des Programms nicht.

(C) **KBD** heißt »keyboard«, auf deutsch »Tastatur«. Es wird behandelt wie eine Datei (davon später mehr). Wir lesen also mit **Read** aus der Tastatur ein Zeichen in die Variable »ch« ein.

(D) Ist dieses Zeichen in der Menge »ziffern«, die wir vorhin definiert haben, enthalten, dann wird es mit dem folgenden **BEGIN/END**-Block weiterverarbeitet. Wenn nicht, wird das Programm bei (H) fortgesetzt.

(E) **NOT, OR** und **AND** sind Operatoren, die für den Rechner logische beziehungsweise bitweise Verknüpfungen sind.

Dazu holen wir etwas weiter aus:

NOT also nicht, kehrt immer den logischen Wert um. Verknüpft man es mit der **IF**-Anweisung, dann wird immer dann zur Ausführung mit **THEN** geschritten, wenn die genannte Bedingung nicht zutrifft. Beispiel:

```
IF x=6 THEN Write('sechs');
```

erzeugt den Ausdruck »sechs«, wenn die Variable x auch tatsächlich mit dem Wert 6 belegt ist.

```
IF NOT x=6 THEN Write('anders');
```

erzeugt immer dann den Ausdruck 'anders', wenn die Variable x eben nicht 6 als Wert enthält.

AND verknüpft zwei Bedingungen nach dem Motto »und zugleich«. Es ist nur dann die Aussage wahr, wenn jede Einzelbedingung wahr ist. Beispiel:

```
IF (x>5) AND (x<10) THEN Write('zwischen 5 und 10');
```

erzeugt keinen Ausdruck, wenn der Variablenwert nicht zwischen 5 und 10 liegt.

OR verknüpft zwei Bedingungen im Sinne von »oder aber auch«. Das heißt, es muß nur eine Bedingung richtig (wahr) sein, dann gilt die gesamte Verknüpfung als wahr. Beispiel:

```
IF (x<5) OR (x>10) THEN
Write('nicht im Bereich von 5 bis 10');
```

Zurück zu unserer Eingaberoutine. Was wollten wir überhaupt? Nach der zweiten und der vierten Stelle der Koordinatenziffern darf keine Ziffer zugelassen werden, die größer ist als 5. Genau das bewirkt die Anweisung bei (*E*). Schauen Sie sich bitte diese Stelle gründlich an. Wichtig sind auch die Klammern. Dabei gilt: Die Verknüpfung **AND** ist immer enger als die Verknüpfung **OR**. Man kann dies mit der »Punkt-vor-Strich«-Regel aus dem Rechnen vergleichen. Hier gilt entsprechend: Wenn Klammern nichts anderes vorschreiben, wird zuerst mit **AND** und danach mit **OR** verknüpft.

Das mag vorläufig genügen. Wir wollen uns den Turbo-Spaß nicht mit unnötig viel Theorie verderben.

Sind nun alle Bedingungen der Zeile (E) erfüllt, z.B. wenn für die vierte Ziffer eine »2« getippt wurde, dann wird der Zähler z, der alle Ziffern und nur diese mitzählt, um eins erhöht, und das Zeichen wird auf dem Bildschirm ausgegeben.

Damit haben wir auch gleich einen Unterschied kennengelernt:

Ein einfaches **Read(ch)** würde das eingetippte Zeichen sofort auf dem Schirm mitschreiben.

Read(KBD,ch) dagegen hält sich da zurück. Es belegt zwar ebenfalls die Variable »ch«, jedoch ohne optische Unterstützung.

(F) Noch einmal zur Verknüpfung **OR**:

Wenn es sich soeben um die vierte Ziffer gehandelt hat, aber auch wenn es die zweite gewesen wäre, startet ein weiterer **BEGIN/END**-Block, der die Aufgabe hat, einen Trennpunkt in die Koordinaten einzufügen, sowohl in der Variablen »grin« als auch auf dem Schirm.

(G) Das ganze Spielchen läuft so lange, bis die Taste **<RETURN>** gedrückt wird (für den Computer ist das **<CTRL/M>**) oder bis »grin« eine Länge von 8 Zeichen angenommen hat.

Ist eines von beiden der Fall (schon wieder ein **OR**!), dann ist hiermit die Prozedur am Ende angelangt, was im Programm mit **END** und einem Strichpunkt dahinter vermerkt ist.

Unbemerkt wurde gleichzeitig mit dem Aufbau der Variablen »grin« auch die Variable »breite« mitverändert. Sie hat exakt den gleichen Wert wie der formale Prozedurparameter und wird auf diese Weise ins Hauptprogramm übernommen (zurückgeschrieben). Dort geben wir sie zur Kontrolle in der Mitte des frisch geputzten Bildschirms aus.

Zur Form:

- Parameter werden hinter dem Prozedurnamen aufgeführt und müssen vom Typ her auch dort definiert werden. Die Typdefinition muß im Deklarierungsteil vorgenommen werden, wenn es sich nicht um Standard-Datentypen wie Integer usw. handelt. Im Kopf der Prozeduranweisung darf nur der Typbezeichner stehen.

- Bedingungen für die Verknüpfungen **AND**, **OR** und **NOT** müssen in Klammern geschrieben werden. Geschachtelte Klammern sind zulässig.

- Bauen Sie die Prozeduren genauso sorgfältig und optisch klar auf, wie wir das oben schon besprochen haben. Es lohnt sich. Leerzeilen erhöhen die Lesbarkeit. Kommentare sollten vor dem Beginn eines Abschnitts stehen.

Aufgaben:

1. In Programm P5 haben wir zwei Austauschzeilen vorgesehen. Werfen Sie diese hinaus, und bauen Sie statt dessen die Prozeduraufrufe »gradin(breite[n]);« und »gradin(laenge[n]);« ein. Setzen Sie nun den Pascal-Text dieser Prozedur vor das Wort **BEGIN** des Hauptprogramms, und starten Sie zum Probelauf!

2. Speichern Sie die Prozedur als GRADIN.PD auf Diskette ab. Wir brauchen sie später wieder.

3. Bauen Sie nun das Programm P5 mit der Prozedur »gradin« so um, daß Gradzahlen von -180.00.00 bis +180.00.00 angenommen werden.

4.6. Mehrere Parameter, Copy, Val, Length, Formate, Backspace Programm P7: Eingabe von Realzahlen mit PROCEDURE »realin.pd«

Wenn Sie das vorhergehende Programm P6 ausprobiert haben, werden Sie gemerkt haben, daß es noch einen gewaltigen Mangel hat: Fehlerhafte Eingaben lassen sich nicht mehr korrigieren. Wir verbessern unsere Technik in dieser Beziehung, wollen aber nicht noch einmal mit der Koordinatenform arbeiten, sondern erproben unsere Kräfte an der Eingabe von Realzahlen, wobei wir uns zum Ziel setzen, nur eine ganz bestimmte Anzahl Ziffern zuzulassen. Aber auch fehlerhafte, nicht gewollte Eingaben müssen wieder rückgängig gemacht werden können.

Bei dieser Gelegenheit ist anzumerken, daß eine Pascal-Prozedur allein nicht lauffähig ist. Der Compiler merkt bei den Parametern sofort, daß keine entsprechenden Partnervariablen im Hauptprogramm vorhanden sind.

Übrigens sind die Pascal-Anweisungen **Read**, **Write** usw. ebenfalls Prozeduren, die aber bereits im Pascal-System existieren, so daß sie nur noch mit ihrem Namen und den entsprechenden Parametern aufgerufen werden müssen.

Hier nun die Prozedur »realin« ohne das aufrufende Hauptprogramm.

```
(*A*)  PROCEDURE realin (VAR realin:Real;stellen:Integer);
       VAR
         line:String[20];
         y:Integer;
         ch:Char;
         zeichen:Set Of Char;
       BEGIN
         zeichen:=['0'..'9','-','+','.'];
(*B*)    line:='';
         REPEAT
           Read(KBD,ch);
(*C*)      IF ch IN zeichen THEN
           BEGIN
              IF Length(line) < stellen THEN
             BEGIN
               Write(ch);
               line:=line+ch;
             END;
           END;
```

```
(*D* )      IF (ch=#127) AND (Length(line) > 0 ) THEN
            BEGIN
(*E*)         Write (ch,' ',^H);
              line:= Copy (line,1,Length(line)-1);
            END;
(*F*)     UNTIL ch=^M;
          VAL (line,realin,y);
        END;
```

Erläuterungen:

(A) Diesmal verwenden wir zwei Parameter: Der Variablenparameter »realin« soll die gewünschte Zahl aufnehmen, der Festwertparameter »stellen« muß die gewünschte maximale Länge aus dem Hauptprogramm übernehmen. In dieser Stringlänge sind auch ein eventueller Dezimalpunkt und ein eventuelles Vorzeichen enthalten.

Der Parameter »stellen«, in dem wir die Stringlänge aufgreifen, ist diesmal eigentlich keine Variable, sondern ein sogenannter Wertparameter. Doch darüber unterhalten wir uns im nächsten Abschnitt ausführlicher (siehe 4.7).

(B) Die Variable »line« verwenden wir, um alle eingegebenen Zeichen zunächst als String zu erfassen. Zum Programmstart wird sie vorsichtshalber als Leerstring definiert (Initialisierung), denn Turbo vergißt so schnell keine alten Belegungen.

(C) In der **REPEAT**-Schleife verwenden wir die Variable »zeichen«, die vom Typ einer Menge ist. Wir haben darin die Menge aller zulässigen Zeichen abgelegt.

(D) Wenn die Taste> **<DEL>** gedrückt wurde, erkennt der Computer dies als CTRL/H (Backspace). Wir berücksichtigen auch dieses Zeichen, damit wir beim Eintippen der Zahl ein eventuell fehlerhaftes Zeichen wieder löschen können. Das darf aber nur dann zugelassen sein, wenn der String immer eine größere Länge hat als Null; denn sonst rutschen wir über unsere linke Stelle ins Leere.

(E) Und jetzt heißt es aufpassen und genau verfolgen, was der Cursor auf dem Bildschirm anstellt. Da er unsichtbar bleibt, ist das besonders knifflig:

Mit **Write**(ch) wird im Falle des Korrigierens das Backspace ausgelöst. Unser Cursor steht jetzt auf dem zu löschenden Zeichen.

Mit **Write**(' '), also einem Blank (Leerzeichen), löschen wir das fehlerhafte Zeichen. Jetzt steht der Cursor aber wieder ein Zeichen zu weit rechts.

Also Kommando zurück mit einem erneuten CTRL/H (>Backspace) auf die alte Position!

Was wir auf diese Weise am Bildschirm bereinigt haben, müssen wir auch unserer Variablen »line« bzw. deren Inhalt antun.

Der **Copy**-Befehl übernimmt das: Wir belegen die Variable »line« mit einem Teil ihrer selbst, nämlich mit allen Zeichen außer dem letzten.

Beispiel:

Eingabe: 25.56 steht auf dem Schirm. Die Länge ist demnach 5. Nach dem Löschen der letzten Ziffer sieht das so aus: 25.5 auf dem Schirm mit einer Länge von 4 Zeichen. In »line« steht aber noch 25.56 mit der Stringlänge 5.

Daraus kopieren wir vier Zeichen (eins weniger als die Länge angibt), von vorn gezählt.

Length(line) ermittelt dabei die Länge des Strings »line«.

(F) Wenn das Abschlußzeichen ^M von **<RETURN>** empfangen wurde, kann zur Auswertung der Zahl übergegangen werden, denn mit Strings kann man nicht rechnen, auch wenn sie noch so schön nach einer Zahl aussehen.

Der entsprechende Befehl heißt zwar wie in BASIC ebenfalls **Val**, aber in Pascal müssen dahinter in Klammern zunächst der auszuwertende String, dann die Realzahlvariable, die den Wert aufnehmen soll, und außerdem eine Kontrollvariable aufgeführt werden.

Letztere nimmt immer den Wert Null an, wenn alles glattgegangen ist. Ansonsten gibt sie die Nummer desjenigen Zeichens aus dem String an, das sich nicht auswerten ließ. Vorteil: Das Programm bricht nicht wegen eines nicht auswertbaren Zeichens ab.

Wie immer schließt die Prozedur mit einem strichpunktbehafteten **END**.

Speichern Sie nun dieses nützliche Gebilde unter dem Namen REALIN.PD ab. Sie werden es sicher in Kürze brauchen können, wenn Sie an Turbo dranbleiben.

Zur Form:

- Treten mehrere Parameter auf, so werden sie hinter dem Prozedurnamen in dieselbe Klammer geschrieben. Mit dem Strichpunkt als Trennzeichen kennzeichnet man das Ende einer Parameterart.

- Die vordefinierte Prozedur **Val** verlangt im Gegensatz zu BASIC gleich drei Parameter.

- Stringmanipulationen können mit **Copy** ausgeführt werden, womit man ohne weiteres die BASIC-Befehle MID$, LEFT$ und RIGHT$ nachvollziehen kann.

- Am besten stellt man sich den Befehl **Copy**('ursprung',3,8) so vor: Kopiere aus dem Wort 'Ursprung' mit dem 3. Zeichen beginnend 8 Zeichen. Ergebnis: 'sprung'.

- Werden bei **Copy** mehr Zeichen zum Herauslesen angegeben, als der String enthält, dann wird bis zum Stringende kopiert.

Zum Austesten unserer Prozedur »realin« schreiben wir ein kleines Hauptprogramm:

```
        PROGRAM P7;
        VAR
          zahl:Real;
          stellen:Integer;

        (* Hier muß nun die Prozedur »realin« eingefügt werden:
         ...  ...  ...  ...  ...  *)

        BEGIN (* Beginn des Hauptprogramms *)
          ClrScr;GotoXY(1,13);
          stellen:=5;
(*A*)     Write('Zahl eingeben: ');realin(zahl,stellen);
          GotoXY(1,17);
          WriteLn(zahl:16, '   (Exponentialdarstellung)');
          WriteLn(zahl:5:2 '              (Dezimaldarstellung)');
        END.
```

Erläuterungen:

(A) Hinter dem Prozeduraufruf »realin« stehen die Übergabeparameter in der gleichen Reihenfolge, wie sie von der Prozedur hinter dem Prozedurnamen aufgenommen wurden.

Betrachten Sie einmal die Bildschirmausgaben:

Zuerst wird in einem 16 Zeichen breiten String die eingetippte Zahl rechtsbündig in Exponentialdarstellung gedruckt. Das sollte Ihnen bekannt vorkommen: Bei der Stringausgabe haben wir diese Formatierung kennengelernt.

Wünscht man eine gewöhnliche Dezimaldarstellung, so müssen nach jeweils einem Doppelpunkt hinter der Variablen zusätzlich die Zahl der gewünschten Gesamtstellen sowie die Zahl der gewünschten Nachkommastellen angegeben werden. Gibt man für die Gesamtstellen einen kleineren Wert an als für die Nachkommastellen, dann erfolgt eine linksbündige Ausgabe.

Beispiel:
```
Zahl:=123.567;
Write('Beispiel:',Zahl:0:2);
```
Ausgabe:
```
Beispiel:123.56
```

In jedem Fall wird die gewünschte Zahl der Nachkommastellen berücksichtigt.

Geben Sie in Programm P7 doch mal eine positive Zahl mit vorangestelltem Pluszeichen ein. Wir haben dies ja ausdrücklich zugelassen. Erstaunlicherweise meldet der Rechner den Wert Null für jede dieser Eingaben.

Woran liegt das? Nun, die Prozedur **VAL** ist so konstruiert, daß sie eine Ziffernfolge nur dann auswerten kann, wenn davor und dahinter keine weiteren Zeichen mehr folgen. Eine Ausnahme bildet das Minuszeichen direkt vor Beginn einer Ziffernfolge. Nicht einmal Leerzeichen sind in diesem Fall zulässig. Hier benimmt sich Turbo pingelig. Um dieses Problem für unsere Zwecke zu lösen, bieten sich zwei Möglichkeiten an: Entweder wir lassen das Pluszeichen erst gar nicht zu, oder wir helfen dem Rechner aus der Misere.

Letzteres könnte so aussehen:

Wir führen zunächst eine Auswertung mit **VAL** durch, wie sie in **PROCEDURE** P7 beschrieben ist. Sollte nun ein Pluszeichen gefunden worden sein, dann bricht der Rechner schon bei der ersten Stelle ab und notiert dies in der Kontrollvariablen »y« (siehe oben). Denn auf Pluszeichen ist er ja (leider) nicht eingerichtet. Positiv ist für ihn eine Zahl schon dann, wenn sie kein Vorzeichen trägt. Gehen wir davon aus, daß die weitere Zeichenfolge formgerecht geschrieben wurde, dann können wir den String trotzdem auswerten. Und zwar ab der zweiten Stelle:

```
IF y=1 THEN Val(Copy(line,2,Length(line)),zahl,y);
```

Deshalb fügen wir einfach diese Zeile in Programm P7 und auch in die Prozedur REALIN.PD auf der Diskette ein.

Übrigens braucht nach dem **THEN** in unserer Zusatzzeile kein **BEGIN/ END**-Block folgen, weil es sich hier um eine Einzelanweisung handelt, die zugegebenermaßen etwas komplexer ist. Das macht Turbo nichts aus. Aber sobald Sie diese Anweisung zerlegen, muß das Ganze zwischen **BEGIN** und **END** stehen:

```
IF y=1 THEN
BEGIN
  rest:=Copy(line,2,Length(line));
  Val(rest,zahl,y);
END;
```

Wir setzen hier voraus, daß »rest« ordungsgemäß deklariert ist.

4.7 Wichtiges zu den Prozedur-Parametern Programm P8: Variablen- und Festwertparameter

Wie bereits angedeutet, gibt es zwei Arten von Parametern, mit denen man aus dem übergeordneten (aufrufenden) Programm Werte in eine Prozedur übernehmen kann:

a) die sogenannten *Variablenparameter* (auch *Wechselwertparameter*)
b) die sogenannten *Wertparameter* (auch *Festwertparameter*)

Von welcher Art sie sind, hängt davon ab, wie man sie hinter den Prozedurnamen schreibt. Setzt man das Wort **VAR** davor, so handelt es sich logischerweise um Variablenparameter.

Beispiel: `PROCEDURE test_var (VAR breite, laenge: Integer);`

Läßt man **VAR** weg, wählt man feste Übergabewerte.

Beispiel: `PROCEDURE test_wert (breite, laenge: Integer);`

Auch eine Kombination beider Arten ist möglich. Dann werden Variablen- und Wertparameter säuberlich durch einen Strichpunkt voneinander getrennt. Die Reihenfolge ist gleichgültig. Zum Beispiel:

```
PROCEDURE Test_Var_Wert (VAR flaeche:Integer; breite,laenge:Real);
```

Wie aber machen sich diese Unterschiede beim Programmlauf bemerkbar?

Variablenparameter ändern auch im Hauptprogramm ihren Wert entsprechend den Verarbeitungen ihres Gegenstücks in der Prozedur.

Wertparameter dagegen werden zwar als eine Art Platzhalter übergeben, lassen aber den Wert des Übergabeparameters aus dem Hauptprogramm unverändert, egal, was mit ihren Partnern in der Prozedur alles passiert.

Verdeutlichen wir uns dies an einem Programmbeispiel, das auf zwei verschiedene Arten die Fläche eines Rechtecks berechnet und zusätzlich die Parameter verändert. Zur Kontrolle lassen wir uns am Schluß auch alle Parameter ausgeben.

Probieren Sie Programm P8 ruhig erst einmal aus, bevor wir uns darüber Gedanken machen. Erläuterungen gibt es diesmal nur wenige, weil keine unbekannten Pascal-Begriffe auftauchen.

```
PROGRAM P8;
VAR
  flaeche,breite,laenge:Integer;
PROCEDURE version1 (VAR flaeche:Integer;
                        breite,laenge:Integer);
BEGIN
  flaeche:=laenge * breite;
  flaeche:=flaeche+10;laenge:=laenge+10;breite:=breite+10;
  WriteLn('aus Version1: ':24,flaeche:5,laenge:5,breite:5);
END;
PROCEDURE version2 (VAR laenge,breite:Integer;
                        flaeche:Integer);
BEGIN
  flaeche:=laenge * breite;
  flaeche:=flaeche+10;laenge:=laenge+10;breite:=breite+10;
  WriteLn ('aus Version2: ':24,flaeche:5,laenge:5,breite:5);
END;
BEGIN (* Hauptprogramm *)
  ClrScr;
  laenge:=5;breite:=6;flaeche:=30;
  version1(flaeche,laenge,breite);
  WriteLn (^J,'nach Prozedur Version1: ',
  flaeche:5,laenge:5,breite:5,^J^J);
  flaeche:=30;
  version2(laenge,breite,flaeche);
  WriteLn(^J,'nach Prozedur Version2: ',
  flaeche:5,laenge:5,breite:5);
END.
```

Nach dem Start mit **<R>** erhalten Sie auf dem Schirm folgenden Ausdruck:

```
          aus Version1:    40   16   15
nach Prozedur Version1:    40    5    6

          aus Version2:    40   15   16
nach Prozedur Version2:    30   15   16
```

Es wäre sehr vorteilhaft, wenn Sie sich darüber im klaren wären, wie diese Zahlen zustande kommen: Immer diejenigen Parameter bleiben unverändert, die Sie nicht als Variablenparameter in die Prozedur übernommen haben. Daran ändert auch der gleiche Bezeichner (Name) nichts.

Außerdem erkennen Sie, daß nur die Reihenfolge der Parameter für die Übernahme-Reihenfolge entscheidend ist. Wieder spielt der Bezeichner des Parameters keine Rolle. So wurde z.B. in der ersten Prozedur die Breite als Länge übernommen und umgekehrt. Wie Sie sehen, lassen sich sowohl Zeichenketten als auch Integerzahlen auf die gleiche Weise rechtsbündig formatiert ausgeben.

Aufgaben:

1. Üben Sie an unserem Beispiel weitere Möglichkeiten durch. Verändern Sie dabei auch die Variablennamen und die Parameterbezeichner. Wählen Sie mehrere Parameter usw.

2. Ändern Sie das Programm P8 so um, daß Sie damit das Volumen eines Quaders (Kegels, Zylinders oder ähnlichem) berechnen können. Die Zahl PI steht Ihnen zur Verfügung (Turbo macht's möglich). Wir werden auch darauf später wieder zurückkommen.

Zusammenfassung und Ergänzungen:

- Prozeduren sind Pascal-Unterprogramme, die mit Hilfe von Parametern Daten aus dem Hauptprogramm oder einer aufrufenden Prozedur übernehmen, weiterverarbeiten und wieder zurückgeben können.

- Wir unterscheiden Variablenparameter, die aus dem Hauptprogramm einen Variablenwert zur Weiterverarbeitung in eine Prozedur übernehmen und diesen als neues Ergebnis dem Hauptprogramm zurückreichen, und Festwertparameter, die

lediglich der Prozedur einen Wert zur Verarbeitung übergeben. Das ist aber eine Einbahnstraße, denn nach Rückkehr ins Hauptprogramm wird er dort unverändert wieder aufgefunden.

- Jeder sogenannte aktuelle Parameter aus dem übergeordneten Programm muß als Partner einen sogenannten formalen Parameter im Prozedurkopf vorfinden. Die Reihenfolge wird unabhängig von irgendwelchen Namensbezeichungen streng eingehalten.

- Die Datentypen der Parameter werden im Prozedurkopf durch vordefinierte Typen deklariert. Auf den Unterschied zwischen globalen und lokalen Bezeichnern wird noch einmal hingewiesen!

- Prozeduren stehen im Deklarierungsteil eines Programmtextes. Sie werden unter dem Namen aufgerufen, unter dem sie definiert wurden.

- Will man in einem Programm mit vielen Unterprogrammen die Übersicht behalten, so kann man auch zunächst nur die Prozedurköpfe mit den Parameterlisten schreiben und den Programmblock irgendwo später im Deklarierungsteil unterbringen. Dazu wird im ursprünglichen Programmkopf der Zusatz **FORWARD** angehängt. Der Programmtext beginnt später nur mit dem Namen des Unterprogramms.

Ein Beispiel soll dies veranschaulichen. Wir haben uns für diese **FORWARD**-Demonstration an das Programm P8 angelehnt:

```
        PROGRAM P8_Forward_Demonstration;   (* P8_FORWD *)
        VAR
          flaeche,laenge:Real;
        PROCEDURE fl1 (VAR quadratflaeche:Real;laenge:Real);FORWARD;
        PROCEDURE fl2 (VAR kreisflaeche:Real;laenge:Real);FORWARD;
        PROCEDURE fl1;
        BEGIN
          flaeche:=Sqr(laenge);
        END;
        PROCEDURE fl2;
        BEGIN
(*A*)     flaeche:=Pi*Sqr(laenge/2);
        END;
        BEGIN (* Hauptprogramm *)
          ClrScr;
          WriteLn('Flaechenberechnung f. Quadrat und Kreis'^J^J^M);
          Write('Grundmass in m:  '); ReadLn(laenge);
          fl1(flaeche,laenge);
```

```
    WriteLn(^J^J'Quadratflaeche in Quadratmeter:  ',
    flaeche:8:2,^J);
    fl2(flaeche,laenge);
    WriteLn(^J'Kreisflaeche in Quadratmeter:    ',
    flaeche:8:2);
  END.
```

Erläuterung:

(A) Die Funktion **Sqr**(zahl) bildet die Quadratzahl aus einer beliebigen Zahl.

Kapitel

5

Turbo-Prozeduren von Diskette

5.1 Der Befehl $I

Turbo hat eine sehr geschickte Art, mit Prozeduren umzugehen. Denn wie wir vorhin gesehen haben, stört jedes Unterprogramm den Überblick über das eigentliche Hauptprogramm. Warum soll man eigentlich alle Prozeduren während eines langen Programms ständig mitschleppen, wenn man sie auch auf eine Diskette ausquartieren kann?

Genau das läßt sich sehr elegant verwirklichen. Man kann nämlich sein Hauptprogramm so schreiben, als seien die aufzurufenden Prozeduren bereits Pascal-eigene Standardprozeduren. Das ist aber nichts Neues für uns. Neu ist nur, daß diese selber erschaffenen Prozeduren gar nicht im Programmtext stehen müssen. Es ist lediglich ein Hinweis unterzubringen, daß die Prozedur auf Diskette existiert.

Der Compiler, der aus dem Pascal-Text ein lauffähiges Programm erstellt, holt sich über unsere Diskettenstation automatisch die angegebenen Unterprogramme und bindet sie an der vorgesehenen Stelle ein. Den Befehl dazu erkennt er an einer Kommentarzeile,

die entweder mit geschweiften Klammern oder mit runden Klammern und Sternchen begrenzt ist. Dahinter stehen die Zeichen $ und I sowie der Dateiname, unter dem das Programm auf Diskette abgelegt wurde.

Also zum Beispiel `{$I GradIn.P10 }`

oder `(*$I GradIn.P10 *)`

Zur Form:

Achtung! Die Zeichenfolge $I muß unmittelbar hinter der Kommentarklammer folgen. Ein Leerzeichen nach »Klammer auf« führt zum Ignorieren der Include-Anweisung.

Programmteile, die auf diese Weise übernommen werden, tragen den Namen »Include-Dateien«. Dieses Wort kommt aus dem Englischen und heißt include (einschließen, einbinden).

Folgendes Schema hilft Ihnen sicher, diese Art der Programmierung zu verstehen:

1. Deklarierungsteil des Hauptprogramms

2.
```
(*$I unterprogramm1 *)
(*$I unterprogramm2 *)
.... .... .... ....
```

3. Anweisungsteil des Hauptprogramms
```
BEGIN
.... ...
unterprogramm1;
.... ...
unterprogramm2;
.... ...
END.
```

Wir hoffen, Sie wagen sich mit uns an ein etwas größeres Programmbeispiel, das aber durch die Verwendung von Include-Dateien viel von seiner sonst vorhandenen Unübersichtlichkeit verliert.

5.2 Verwendung von Include-Dateien

Unser Ziel haben wir noch nicht aus den Augen verloren: Wir wollten mit Koordinaten rechnen, um uns die Entfernung zwischen zwei beliebigen Punkten ausgeben zu lassen. Wir kommen dem schon näher, wenn es uns gelingt, alle Koordinaten in Dezimalschreibweise darzustellen.

Warum dies notwendig ist? Es gilt nämlich: 11.30.00 Grad sind 11,5 Grad und eben nicht 11,3 Grad, weil 30 Minuten ein halbes Grad sind.

Für unsere Rechnerei müssen wir wissen – und dies auch nachher dem Computer beibringen – , daß 1 Minute ein sechzigstel Grad und eine Sekunde ein dreitausendsechshundertstel Grad ist. Richtig! Wie wir das von Stunden, Minuten, Sekunden her gewöhnt sein sollten.

Dann kann ein Programm dazu doch gar nicht so schwierig zu erstellen sein, wenn wir uns zunächst einmal darauf beschränken, den Unterschied zwischen zwei geographischen Breitenangaben zu berechnen.

Eine Prozedur aus Kapitel 4 haben Sie ja sicher schon auf der Diskette, wenn nicht, holen Sie das bitte nach:

Es ist »gradin« aus dem Programm P6, die Grad-Eingabe-Prozedur. Sie haben sie vielleicht schon erweitert und perfektioniert. Denn hier versteht Turbo überhaupt keinen Spaß: Wenn die einzubindende Prozedur nicht fehlerfrei ist, dann wirft der Compiler das Hauptprogramm zum Arbeitsspeicher hinaus. Er ist aber so zuvorkommend, daß er es vorher noch schnell auf Diskette abspeichert. Anschließend holt er das verbesserungswürdige Unterprogramm herein, so daß Sie sofort auf die Fehlersuche gehen können.

Sind Sie schon soweit, daß Sie das folgende Programm mit etwas weniger Kommentar analysieren können?

5.2.1 Das Rumpfprogramm P9: Koordinaten

```
      PROGRAM P9;
(*A*) TYPE
        grad=String[8];
      VAR
        breite1,breite2,differenzG:grad;
        breite1d,breite2d,differenzD:Real;
```

```
(*B*) (* Include-Prozeduren: *)
      {$I GradIn.PD}
      {$I Grad_Dez.P10}
      {$I Dez_Grad.P11}
      BEGIN (* Hauptprogramm *)
        ClrScr;
(*C*)   Write('geographische Breite vom 1.Ort: ');
        gradin(breite1);
        Write(^J^M,'geographische Breite vom 2.Ort: ');
        gradin(breite2);
       (* Berechnung der Differenz der Breiten *)
(*D*)   grad_dez(breite1,breite1d);
        grad_dez(breite2,breite2d);
(*E*)   differenzD:=Abs(breite1d-breite2d);
(*F*)   dez_grad(differenzD,differenzG);
        (* Bildschirmausgabe der Ergebnisse: *)
(*G*)   WriteLn(^J^J^M,'Breitendifferenz zw. Ort 1 und Ort 2: ');
        Write(differenzD:8:4,' Grad');
        Write(' = ', differenzG, ' Grad.Min.Sek.');
      END.
```

Erläuterungen:

(A) Im Kopf des Hauptprogramms werden alle Variablen definiert, die globale Geltung haben sollen. Wir benötigen je eine Variable für die geographischen Breiten des ersten und des zweiten Ortes, und zwar jeweils in Dezimaldarstellung und in Graddarstellung. Auch die Differenz dieser Werte wollen wir in beiden Formen behandeln. Deshalb jeweils der Buchstabe G für Graddarstellung oder D für Dezimalzahlangabe als Zusatz.

(B) Hier werden die benötigten Prozeduren vom Compiler aufgerufen. Wenn sie auf Diskette vorhanden sind und fehlerfrei eingebunden werden können, werden auch die nachfolgenden Anweisungen angenommen.

(C) Damit wir wissen, was überhaupt eingegeben werden soll, muß auf dem Schirm ein Hinweis erscheinen. Danach kann jeweils die hausgemachte Prozedur aufgerufen werden. Wir geben ihr immer einen Parameter mit, der die Koordinaten zunächst als String aufnehmen soll.

(D) Zum Rechnen müssen (oder wollen) wir die Koordinaten in Dezimalform umwandeln. Das kann das Unterprogramm »grad_dez.P10«. Als Parameter benötigt es unbedingt irgendwelche Koordinaten in Stringform, die die übliche Schreibweise gg.mm.ss aufweisen müssen. Als weiteren Parameter finden wir

eine Realzahlvariable, in die wir uns nach dem Prozedurlauf die Koordinate in einer Dezimalzahl zurückschreiben lassen.

(E) Die Differenz aus den beiden eingetippten Koordinaten wird gebildet.

(F) Etwas schwieriger als die eben durchgeführte Operation ist es, die Differenz, die ja ebenfalls in einer Dezimalzahl ausgerechnet wurde, wieder in die Koordinatenform mit Grad.Min.Sek. zurückzuverwandeln. Aber die Prozedur »dez_grad« schafft das schon. Wir übergeben ihr die Differenz im Dezimalsystem und erwarten von ihr, daß sie uns einen geordneten String abliefert.

(G) Damit wir unseren Rechner auch kontrollieren können, lassen wir ihn die Ergebnisse auf den Bildschirm schreiben.

Zu Form und Programmierarbeit:

- Weil wir gerade beim Stichwort Kontrolle sind: Wenn Sie beim Testlauf Ihres Programms Ergebnisse erhalten, die mit Ihren Vorstellungen und mit Ihrem Taschenrechner nicht übereinstimmen, ist es zweckmäßig, nicht ziellos im Programm herumzustochern, um nach langem Suchen und immer lauter werdendem Fluchen vielleicht doch noch den Fehler zu finden, sondern, beginnend mit den ersten Ergebnissen, Kontrollausdrucke in den Text zu programmieren. Wir hatten hier deren sechs eingebaut, die wir aus Platzgründen wieder entfernt haben. Und sie waren auch wirklich notwendig. Denken Sie bitte nicht, daß unsere Programme immer auf Anhieb gelaufen wären!

- So wie unser Programm P9 jetzt beschaffen ist, können Sie es aber noch nicht brauchen, denn Turbo kennt die Anweisungen »grad_dez« und »dez_grad« nicht, und auf Diskette haben wir diese Prozeduren auch noch nicht. Aber was noch nicht ist, kann ja im nächsten Abschnitt werden.

- Wir haben der deutlicheren Schreibweise wegen die Bezeichner »grad_dez« und »dez_grad« mit einem Strich auseinandergezogen. Das ist gefährlich, denn hier darf (auf keinen Fall!) weder ein Leerzeichen noch ein »-« verwendet werden. Bei einem Zwischenraum würde nämlich der Name nur aus dem ersten Teil bestehen, weil Blanks eben Trennzeichen sind. Die zweite unerlaubte Schreibweise wäre die Verwendung eines Verknüpfungszeichens (Operationszeichen oder kurz Operator). Dazu gehören »-«, »+«, »*«, »/«, »>«, »<« und »=«. Da auch Klammern aller Art in Pascal ihre eigene Bedeutung haben, bleibt außer dem Unterstreichungszeichen »_« nicht mehr viel übrig. Schauen Sie sich mal die Tastatur an.

- Ein weiteres Kontrollzeichen bzw. eine Kontrollzeichen-Kombination haben wir verwendet: ^J^M. Sie bewirkt einen Zeilensprung und setzt den Cursor an die erste Bildschirmspalte. Damit kennen wir mit ^H schon drei solcher Cursorsteuerungs-Möglichkeiten. Sie werden immer mit einer **Write**- oder **WriteLn**-Anweisung ausgegeben, brauchen aber nicht mit Kommata getrennt zu werden, wenn man mehrere hintereinander verwenden will.

So, jetzt fehlen uns nur noch die beiden Prozeduren, die die Umformungen der beiden Koordinatenschreibweisen bewirken. Wir wollten Ihnen dies eigentlich als Aufgabe stellen. Die erste Umformung von Grad.Min.Sek. in Grad,xxx ist recht einfach. Sie sollten sie, möglichst ohne in 5.2.2 nachzulesen, einmal selbst versuchen. Die zweite Umformung hat etliche Tücken. Deswegen werden wir sie Ihnen nicht unvorbereitet zumuten.

Doch eins müssen wir hier anmerken: Es gibt viele Möglichkeiten, zum Ziel zu kommen. Wir haben absichtlich nicht immer die raffinierteste gewählt, denn wir wollen unser Hauptaugenmerk auf das Verständnis von Turbo-Pascal legen. Wenn Sie in Abschnitt 5.2.3 an irgendwelchen Stellen meinen, das gehe doch auch viel einfacher, dann wären wir überhaupt nicht überrascht. Sie sollten aber Ihre eigene Version auch wirklich ausprobieren, denn oft hat man sich ein bißchen zu früh gefreut. Erst der einwandfreie Programmlauf gibt Ihnen recht. Meckern allein gilt nicht!

5.2.2 Umwandlung von Koordinatenform in Dezimalform Prozedur P10 »grad_dez«: Val, Copy

Das Verfahren ist recht einfach: Die eingetippten Koordinaten werden von der Prozedur als Festwertparameter (kurz Wertparameter) aufgenommen, und zwar immer in einem acht Zeichen langen String. Nennen wir ihn einfach »koo«.

Zeichen 1 und 2 enthalten die vollen Gradzahlen, Zeichen 4 und 5 die Minuten und Zeichen 7 und 8 die Sekunden. Die Punkte stehen bei 3 und 6 und werden hier nicht benötigt. Der **Copy**-Befehl holt uns die gewünschten Teile heraus, die **Val**-Routine wandelt die kurzen Zeichenketten in Ganzzahlenwerte um. Wir haben diesmal beide Befehle geschachtelt, um etwas Platz zu sparen.

Um die Koordinaten als Dezimalzahlen ausgeben zu können, rechnen wir die Untereinheiten mit den oben bereits genannten Faktoren 1/60 bzw. 1/3600 um. Fertig ist dieses Unterprogramm, wenn Sie es mit **END** und dem Strichpunkt abschließen. Speichern Sie es nun unter dem Namen »grad_dez.P10« ab. Ausprobieren läßt sich diese Prozedur allerdings zum jetzigen Zeitpunkt noch nicht.

Später wird der Variablenparameter (oder auch Wechselwertparameter) »kooD« vom Hauptprogramm übernommen.

```
PROCEDURE grad_dez (koo:grad;VAR kooD:Real);
VAR
  grad,min,sec,y:Integer;
BEGIN
  kooD:=0;
  Val(Copy(koo,1,2),grad,y);
  Val(Copy(koo,4,2),min,y);
  Val(Copy(koo,7,2),sec,y);
  kooD:=grad + min/60 + sec/3600;
END;
```

5.2.3 Umwandlung von Dezimalgrad in Grad.Min.Sek. Prozedur P11 »dez_grad«: Int, Frac, Round, Val, Str

Jetzt wird es zugegebenermaßen etwas komplizierter: Wir rechnen eine Dezimalzahl in Grad, Minuten und Sekunden um. Das Ganze verfolgen wir an einem Zahlenbeispiel. Die gegebene Gradzahl sei 10.061111..., also eine periodische Dezimalzahl.

Falls Sie ohne unsere Erklärungen klarkommen, hier ist zunächst der Pascal-Text zur geforderten Include-Datei »dez_grad.P11«:

```
(*A*) PROCEDURE dez_grad (kooD:Real;VAR kooG:grad);
      VAR
(*B*)   zweistell:String[2];
        zahl,dez:Real;
        y:Integer;
      BEGIN
(*C*)   kooG:='';
(*D*)   dez:=Int(kooD);
        Str(Int(dez):2:0,kooG);kooG:=kooG+'.';
(*E*)   dez:=Frac(kooD);dez:=Round(100*dez*60)/100;
                          (* dez:=dez*60; *)
        Str(Int(dez):2:0,zweistell);
(*F*)   IF Copy(zweistell,1,1)=' '
        THEN zweistell:='0'+Copy(zweistell,2,1);
        Val(zweistell,zahl,y);
        IF zahl < 10
        THEN kooG:=kooG+'0'+Copy(zweistell,2,1)+'.'
        ELSE kooG:=kooG+zweistell+'.';
(*G*)   dez:=Frac(dez);dez:=(dez*60);
                         (* dez:=Round(dez*60); *)
```

```
  Str(dez:2:0,zweistell);
  IF Copy(zweistell,1,1)=' ' THEN
         zweistell:='0'+Copy(zweistell,2,1);
  Val(zweistell,zahl,y);
   IF zahl < 10
  THEN kooG:=kooG+'0'+Copy(zweistell,2,1)
  ELSE kooG:=kooG+zweistell;
END;
```

Erläuterungen:

(A) Die Zahl 10.06111... kommt als Festwert in der Variablen »kooD« im Kopf der Prozedur (als Realzahl) an. »kooG« wird später die Koordinatenform enthalten.

(B) Für unsere Operationen benötigen wir einen zweistelligen String, eine Variable »zahl« für bestimmte Überprüfungen, eine veränderliche Realzahl »dez«, mit der die Zerlegung der gegebenen Zahl bewerkstelligt wird, und die Kontrollvariable »y« für die Standardprozedur **Val**. Wir kennen sie ja schon.

(C) Wir sorgen dafür, daß unser Ergebnis auch in einer »sauberen« Variablen abgelegt wird.

(D) **Erster Schritt:**

Wir bilden den Integer-(Ganzzahl-)Wert von 10.06111.... Das Ergebnis ist 10.000000..., also eine Realzahl.

Davon können wir nur die ersten beiden Stellen gebrauchen, die wir in einen String umwandeln. Setzen wir noch einen Trennpunkt dazu, dann ist unser erstes Zwischenergebnis für »kooG« schon recht vorzeigenswert: »10.« .

(E) **Zweiter Schritt:**

Wir bilden mit **Frac** den Nachkommateil unserer gegebenen Zahl. Das macht .06111... für die Belegung von »dez«. Das sind die Sechzigstel eines vollen Grades. Mal 60 macht 3.666....Minuten. Runden wir das auf zwei Stellen, dann sieht es so aus: 3.66 Minuten.

Davon können wir wiederum nur den ganzzahligen Teil gebrauchen, müssen aber damit rechnen, daß auch zweistellige Minuten vorkommen. »zweistell« jedenfalls ist in unserem Beispiel »3«.

(F) **Dritter Schritt:**

Das sieht zunächst einmal verwirrend aus. Es ist nämlich möglich, daß die Minutenanzahl nur aus einer Stelle besteht. Dann sind wir gezwungen, die Form zu wahren und eine zweite Stelle, nämlich die Null, voranzustellen.

Prüfen wir deshalb, ob die Minutenzahl einstellig ist, also zwischen null und zehn liegt. Das ist in unserem Beispiel tatsächlich so. Also tritt diese Bedingung in Kraft, d.h., die dahinterliegende Anweisung wird durchgeführt: Unsere Variable »kooG«, die bisher den Inhalt »10.« hatte, wird erweitert um »0« und den Inhalt von »zweistell:2:1«, also »3«. Dahinter setzen wir wieder den Trennpunkt.

Warum wir ausgerechnet »zweistell:2:1« genommen haben? Weil »zweistell« auch wirklich immer ein zweistelliger String ist. Und wenn der Zahlenwert z.B. 1 ist, dann hat »zweistell« ein Leerzeichen voraus. Deshalb verwenden wir nur einen Teilstring davon, nämlich von Zeichen Nummer zwei ab genau ein Zeichen.

»kooG« sieht jetzt schon fast fertig aus: »10.03.«

(G) »dez« hat immer noch den Wert 3.66. Der Nachkommateil ist bereits der gerundete Sekundenwert, wenn wir ihn mit 60 multiplizieren. Das macht 39.6. Die Integerzahl davon ist 39, gerundet).

Der Rest ist fast der gleiche wie bei (F), lediglich der Trennpunkt am Schluß wird überflüssig.

Ergebnis: »kooG« ist »10.03.40«. Und das dürfen wir ruhigen Gewissens wieder dem Hauptprogramm zurückgeben.

Wir haben bei den Rundungen eine Alternative ausprobiert. Sie steht in Kommentarklammern bei (E) und (G). Sie hat nur den Fehler, daß sie manchmal statt 10.20.00 lieber 10.19.60 schreibt. Das ist zwar mathematisch nicht falsch, aber formal falsch. Sie sehen, auf was man alles achten muß!

Zur Form:

- Die Anweisung **IF** ..(1).. **THEN** ..(2).. kann auch mit **ELSE** ..(3).. verknüpft werden, was sich übersetzen läßt mit »Wenn Bedingung (1) erfüllt ist, dann führe Anweisung

(2) aus, ansonsten führe Anweisung (3) aus«. Auch hier gilt, daß nach **ELSE** genau so wie nach **THEN** ein **BEGIN/END**-Block folgen muß, wenn mehr als eine Anweisung auszuführen ist.

- **Achtung:** Vor einem **ELSE** darf kein Strichpunkt stehen. Er würde das Ende der **IF**-Anweisung markieren und damit **ELSE** nicht mehr erkennen.

- Wird ein Zahlenwert in einen String verwandelt, so ist zu beachten, daß die Zahl zunächst einmal in Exponentialdarstellung vorliegt. 10.6 sieht also vorerst so aus: 1.0600000000E1.

- Will man Vor- und Nachkommastellen im Dezimalformat, dann muß die Formatieranweisung »:Gesamtstellen:Nachkommastellen« gegeben werden.

- Der **Val**-Befehl liefert falsche Ergebnisse, wenn das erste Zeichen des auszuwertenden Strings ein Leerzeichen ist. Deshalb wird dies bei (*F*) zunächst überprüft und eventuell bereinigt; denn 07 wird z.B. als Wert 7 erkannt.

- Die Integer-Anweisung **Int** liefert keine Integerzahl, wie ihr Name vermuten ließe, sondern eine Realzahl, die normalerweise wieder Exponentialform hat. Das hat tückische Ergebnisse zur Folge, wenn man diese Eigenheit nicht beachtet.

- Rundungen werden von **Int** nicht vorgenommen, es werden nur die Nachkommastellen mit Nullen gefüllt.

- **Round** ist eine Umwandlungsfunktion, die leider nur auf das Runden der ersten Stelle beschränkt ist. Will man auf zwei Stellen nach dem Komma runden, ist so vorzugehen, wie das im Programm unter Punkt (E) beschrieben wurde.

  Eine Rundung auf volle Tausender funktioniert beispielsweise gerade umgekehrt: **Round**(zahl/1000)*1000.

  Beispiel:
  ```
  zahl=27236.33
  zahl/1000= 27.23633
  Round(zahl/1000)=27
  Round(zahl/1000)*1000=27000
  ```

  Schreiben Sie zur Übung eine Prozedur, die man auf eine beliebige Anzahl Nachkommastellen runden kann. Sie sollte die Form haben:

  ```
  roundx(zahl,stellen)
  ```

5.3 Kompilieren mit den Optionen <C> und <M>

Wenn Sie auch die zweite Prozedur unter dem Namen »dez_grad.P11« auf Diskette abspeichern, dann können Sie jetzt das Hauptprogramm P9 als Arbeitsdatei hereinladen und ausprobieren. Starten Sie es mit **<R>**, dann sehen Sie am Schirm die Nummern der kompilierten Zeilen. Sobald ein I erscheint, holt der Compiler die Include-Dateien von Diskette und bindet sie als Programmcode mit in das Hauptprogramm ein.

Gehen Sie nun einmal in das Turbo-Hauptmenü und wählen Sie die Compileroption mit **<O>**. Sie erhalten ein weiteres kleines Menü mit der Frage, in welcher Form Sie denn das Kompilat gern hätten.

Wählen Sie hier die Taste **<C>** und kompilieren Sie anschließend das Programm P9 noch einmal. Wenn Sie nun das Inhaltsverzeichnis betrachten, dann sehen Sie z.B. »P9.COM«. Das ist die eben erzeugte Datei. Sie erhält ein lauffähiges, kompiliertes Programm, das sofort startbereit ist, wenn Sie es von Diskette laden. Turbo hat hier ein Paket Routinen, das sogenannte Runtime-Pack, eingebunden, welches Ihrem Programm von nun an Unabhängigkeit vom Turbo-Compiler garantiert.

Zum Austesten Ihrer Programme sollten Sie aber die Compileroption **<M>** wählen, die das lauffähige Programm zunächst nur im Arbeitsspeicher (deswegen M von Memory) erzeugt.

Welchem Zweck die letzte Option dient, die mit der Taste **<H>** angewählt wird, werden wir im Kapitel 17 besprechen.

Kapitel

6

Funktionen

6.1 Was sind Funktionen?

In der Mathematik versteht man unter Funktionen eindeutige Zuordnungen.

Ein Beispiel:

An einem bestimmten Ort wird ständig die Temperatur gemessen. Dann gilt: Zu jedem Zeitpunkt gehört genau ein Temperaturwert. Die Temperatur ist also eine Funktion der Zeit.

Komplizierter wird es, wenn man aus vielen Einzelwerten, den uns schon bekannten Parametern, erst einen Wert berechnen muß. Beispielsweise die Menge an Kraftstoff, den ein Flugzeug aufnehmen muß, um bei einer bestimmten Wetterlage mit einem bestimmten Abfluggewicht eine bestimmte Strecke in einer bestimmten Höhe mit einer bestimmten Sicherheitsreserve zurücklegen zu können.

Wir haben das Wort »bestimmt« absichtlich jedesmal wiederholt, denn dadurch werden die Werte ausgedrückt, die man erst einmal haben muß, bevor man das Endergebnis ermitteln kann.

Pascal versteht unter **FUNCTION** ein Unterprogramm zur Berechnung eines Wertes aus einem oder mehreren vorgegebenen Parametern.

Im Prinzip sind Turbo-Funktionen Sonderformen der Prozeduren. Jedoch liefert eine Funktion immer genau einen Wert. Die Form unterscheidet sich deshalb etwas von der Prozedurform.

6.2 Aufbau einer Pascal-Funktion Funktion P12: Cos, Sqrt, Abs

Erinnern wir uns: Wir wollten die Länge einer Strecke aus den Koordinaten des Gradnetzes der Erde berechnen. Bisher haben wir gelernt, diese Gradzahlen in einer brauchbaren Form aufzunehmen und beliebig umzuformen. Was hindert uns daran, nun daraus die Entfernung zu berechnen?

Was wir benötigen, ist nur die richtige Formel. Wir verzichten selbstverständlich auf die mathematische Herleitung und geben sie einfach an:

```
Strecke=Erdradius*arccos(cosStr)
```

wobei »cosStr« der Kosinus des Mittelpunktswinkels ist, den die beiden Orte mit dem Mittelpunkt bilden. Und dieser Wert läßt sich natürlich ebenfalls berechnen, und zwar aus den Koordinaten, die wir wegen der Länge der Formel jetzt in Kurzform angeben:

```
geogr. Breite des Ausgangsorts: b1
dto.   Länge                  : l1
geogr. Breite des Zielorts    : b2
dto.   Länge                  : l2
```

Die Formel lautet:

```
cosStr=sin(b1)*sin(b2)+cos(b1)*cos(b2)*cos(abs(l1-l2))
```

Mehr ist nicht notwendig. Wir können unsere Pascal-Funktion schreiben. Sie erhält den Funktionsbezeichner »strecke«.

```
      (* Streckenberechnung Funktion P12.FKN *)
(*A*) FUNCTION arccos(cosstr:Real):Real;
      BEGIN
(*B*)   arccos:=ArcTan(Sqrt(Abs(1-cosstr*cosstr))/cosstr);
      END;

(*C*) FUNCTION strecke (b1,l1,b2,l2:Real):Real;
      VAR
        cosstr: Real;
      CONST
        erdradius = 6378.245;
      BEGIN
        cosstr:=Sin(b1)*Sin(b2)+Cos(b1)*Cos(b2)*Cos(Abs(l1-l2));
(*D*)   strecke:=erdradius*arccos(cosStr);
      END;
      (* Streckenberechnung Funktion P12.FKN *)
```

Wir hoffen, Sie haben soeben gestutzt und gedacht: Das sind doch zwei Funktionen!

Richtig, die Funktion, die aus dem Kosinuswert eines Winkels wieder den Winkel herausrückt, gibt es nämlich auch in Turbo nicht. Das liegt daran, daß Turbo nur die rechnerinternen Umrechnungen verwendet, und die gibt es für Winkel nur über den Tangens. Aber das braucht uns weniger zu interessieren. In jeder Formelsammlung, die etwas auf sich hält, finden wir für die Berechnung des Winkels W aus seinem Kosinuswert Cos(W):

```
arccos(Cos(W))=ArcTan(Sqrt(Abs(1-Cos(W)*Cos(W)))/Cos(W));
```

Wie man unter Turbo diese Formel anwendet, sehen Sie im Programmtext.

Erläuterungen:

(A) Die Kopfzeile einer Funktion sieht der entsprechenden Prozedurzeile ähnlich. Wir erkennen auf jeden Fall schon mal die zu übernehmenden Parameter, im Beispiel ist das ein bereits vorhandener Kosinuswert, eben »cosstr«, eine reelle Zahl.

Der Wert, den die Funktion liefert, trägt immer auch den Namen der Funktion. Das ist praktisch, weil man nicht umdenken muß. Das bedeutet aber auch, daß in der Schlußzeile des Funktionstextes dieser variablenartige Ausdruck mindestens einmal belegt werden muß, damit die Funktion diesen Wert als Ergebnis dem Hauptprogramm übergeben kann.

Neu in der Kopfzeile ist, daß der Typ des Ausdrucks angegeben werden muß, den die Funktion liefern soll. Bei uns ist das mit **Real** erledigt.

(B) Hier erfolgt die Berechnung des Winkels über **ArcTan**, eine Pascal-eigene Funktion.

(C) Diese zweite Funktion nennen wir »strecke«. Sie berechnet aus den Koordinaten die Entfernung und benötigt dazu die Länge des Erdradius. Sie bekommt ihn als Konstante.

Um den Kosinuswert übergeben zu können, benötigen wir die Variable »cosstr«.

(D) Bei der Berechnung wird nun für den Ungeübten sehr unauffällig die Funktion »arccos« aufgerufen. Man erkennt hier nicht, ob es sich um eine bereits vorhandene Pascal-Funktion handelt oder um eine selbst vordefinierte. Jedenfalls ruft eine Funktion hier eine weitere auf. Wir haben zwei Funktionen verschachtelt.

Die Parameterübergabe erfolgt genau wie bei den Prozeduren als (Fest-)Wert- oder als Variablenparameter. Uns genügen hier die Festwertparameter, weil wir nicht die Absicht haben, die Koordinaten mit veränderten Werten dem Hauptprogramm zurückzugeben oder den Kosinuswert »cosstr« zu verunstalten.

Zur Form:

- Jede Funktion muß genau wie jede Prozedur auf jeden Fall vorher – im Deklarierungsteil – definiert werden, bevor man sie aufrufen kann.

- Funktionen können andere Funktionen oder Prozeduren aufrufen und umgekehrt. Etwas heikler wird die ganze Geschichte allerdings, wenn eine Funktion sich selbst aufrufen will. Man spricht dann von Rekursion. Doch davon erst später.

- Der Name der Funktion übernimmt wie eine Variable den Wert, der mit Hilfe der Parameter errechnet wurde.

- Jede Funktion läßt sich in einen Deklarierungs- und einen Anweisungsteil gliedern. Der Anweisungsteil beginnt mit einem **BEGIN** und endet mit einem **END;**.

- Funktionen, die im Deklarierungsteil einer Funktion definiert werden, haben nur in diesem Block Gültigkeit (lokale Funktionen). Will man z.B. die Funktion »arccos«

im gesamten Programm verwenden können, so muß sie direkt hinter dem Deklarierungsteil des Hauptprogramms stehen.

- Das Auslagern von Funktionen aus einem Programmtext geschieht wie bei den Prozeduren oder jedem anderen Programmteil:

  Abspeichern des Textes unter einem Namen – am besten dem Funktionsnamen – auf Diskette oder ähnlichem. An der Stelle, an der die Funktion eingebaut werden soll, erfolgt die Include-Anweisung »$I« mit dem Dateinamen dahinter.

  Wir probieren das gleich an unserem Beispiel aus:

  Sichern Sie den oben erläuterten Pascal-Text unter »strecken.F12«. Den Zusatz ».F12« wählen wir zur Unterscheidung von den ausgelagerten Prozeduren, denen wir ».Pxx« angehängt haben.

6.3 Einbau von Funktionen in das Hauptprogramm Programm P13: Streckenberechnung

Und jetzt ist das Streckenberechnungs-Programm – wenigstens das Kernstück – betriebsbereit, wenn wir das Hauptprogramm noch um einen Schritt erweitern:

```
      PROGRAM P13;   (* Streckenberechnung *)
      TYPE
        grad=String[8];
      VAR
        breite1,breite2,laenge1,laenge2: grad;
        breite1d,breite2d,laenge1d,laenge2d: Real;
        distanz: Real;
        kurs_pd, kurs_fn: Real;

(*A*) {$I GradIn.PD}
      {$I GradRad.P10}
      {$I Strecke.F12}
      BEGIN
        ClrScr;
        WriteLn('Streckenberechnung aus geographischenKoordinaten');
        Write(^J^J^J^M,'Ort 1, Breite: ');
        gradin(breite1);
        grad_dez_rad (breite1, breite1D);
        Write ('   Laenge: ');
        gradin(laenge1);
```

```
    grad_dez_rad (laenge1, laenge1D);
    Write (^J^M'Ort 2, Breite: ');
    gradin(breite2);
    grad_dez_rad (breite2, breite2D);
    Write ('   Laenge: ');
    gradin(laenge2);
    grad_dez_rad (laenge2, laenge2D);
    distanz:= strecke(breite1d,laenge1d,breite2d,laenge2d);
    WriteLn(^J^J^M,'Strecke in km: ',distanz:9:3);
  END.
```

Eine Erläuterung müssen wir noch geben, sonst wundern Sie sich über die Ergebnisse, die dieses Programm liefert.

Erläuterungen:

(A) Wir haben die Prozedur »grad_dez« etwas umgewandelt und unter dem Namen »gradrad.P10« auf der Diskette bereitgestellt. Aufgerufen wird die Prozedur mit »grad_dez_rad«. Sie hat die gleiche Aufgabe wie das ursprüngliche Unterprogramm, enthält jedoch am Schluß noch eine entscheidende Zeile mehr:

```
kooD:=kooD*Pi/180;
```

Damit wird die dezimale Gradzahl auch noch in die Einheit RAD (von Radius) umgewandelt. Denn unser Rechner beherrscht nur diese Einheit, wenn es um Winkel geht.

Für die Umrechnung gilt: 360 Grad = 2*PI. Der Umrechnungsfaktor ist dann gekürzt PI/180. Umgekehrt wandelt man die Winkel aus dem sogenannten Bogenmaß RAD wieder zurück mit dem Faktor 180/PI.

Kurz:

```
Gradzahl=rad*180/Pi
rad=Gradzahl*Pi/180
```

6.4 Ändern einer Include-Datei während des Programmierens: Die Blockbefehle <CTRL/K>-... des Turbo-Editors

Sie werden sich fragen, wie man denn eine Diskettendatei mitten in der Programmierarbeit einfach ändern kann.

Am besten gehen Sie so vor:

<CTRL/Q-C>

1. Gehen Sie mit der Tastenfolge **<CTRL/Q-C>** an das Ende des Pascal-Textes.

<CTRL/K-R>

2. Drücken Sie nun die Tasten **<CTRL/K-R>**, dann erscheint in der ersten Bildschirmzeile die Frage »Lies Block von File: _«.

   Hier tippen Sie den Namen der Diskettendatei ein, die Sie bearbeiten wollen, z.B. »grad_dez.P10«. Turbo fügt nun den neuen Text an das Ende Ihrer bisherigen Programmierzeilen an.

3. Sie verändern wie gewünscht den Text, indem Sie z.B. die folgende Zeile einfügen:

   ```
   kooD:=kooD*Pi/180;
   ```

<CTRL/K-B>

4. Solange Sie keine neuen Markierungen setzen, merkt sich Turbo sowohl Anfang als auch Ende des eingelesenen Blocks. Ansonsten legen Sie den wieder abzulegenden Programmteil fest, indem Sie mit dem Cursor vor den Anfang fahren und die Tasten **<CTRL/K-B>** (für Beginn des Blocks) drücken und das Ende entsprechend mit **<CTRL/K-K>** eingeben. Den gesetzten Block erkennen Sie nun an der reversen Schrift.

<CTRL/K-W>

5. Das Abspeichern geschieht sinngemäß mit **<CTRL/K-W>** (für Write) und der Abfrage des Dateinamens. Wenn Sie denselben Namen wählen wie beim Einlesen, werden Sie erst noch gefragt, ob Sie die alte Datei gleichen Namens überschreiben wollen. Sie ist dann allerdings auf dieser Diskette auch nicht mehr verfügbar.

In unserem Beispiel wählen wir aber einen anderen Namen, der freilich mit dem übereinstimmen muß, der in der $I-Anweisung zu finden ist.

<CTRL/K-H> 6. Wenn Sie z.B. den zusätzlichen Block in Ihrem Programm noch behalten wollen und es stört Sie das reverse Schriftbild, dann tippen Sie **<CTRL/K-H>** (von hide = verstecken). Jetzt unterscheidet er sich äußerlich nicht mehr, ist für Turbo aber dennoch vorhanden, was Sie mit nochmaligem **<CTRL/K-H>** überprüfen können. Diese Tastenfunktion ist also ein Umschalter.

<CTRL/K-Y> 7. Benötigen Sie den Block nicht mehr, kann er mit dem Kommando **<CTRL/K-Y>** vollständig gelöscht werden, natürlich nur im Arbeitsspeicher.

<CTRL/K-V> 8. Wollen Sie ihn aus der unbrauchbaren Lage am Ende des Programms befreien und irgendwo an passender Stelle einbauen, dann bringen Sie den Cursor an die gewünschte Stelle und bedienen die Tasten **<CTRL/K-V>** (wie Verschieben). Jetzt finden Sie Ihren Block am neuen Platz, aber auch nur noch dort.

<CTRL/K-C> 9. Mit **<CTRL/K-C>** können Sie auf die gleiche Weise einen Block einfügen wie mit **<CTRL/K-V>**, der ursprüngliche Block bleibt aber erhalten. Sie erzeugen also eine Kopie.

<CTRL/K-T> 10. Der Vollständigkeit halber sei auch noch der Blockbefehl **<CTR/K-T>** erwähnt. Er markiert eine zusammenhängende Zeichenfolge (ein Wort) als Block, das Sie dann mit den genannten Befehlen ebenso behandeln können.

Diese Blockbefehle wurden praktisch von WordStar übernommen. Man braucht sich also nicht umzustellen, wenn man dieses Textsystem schon kennt. Turbo ist – man sollte es kaum glauben – aber noch etwas komfortabler und vor allem schneller.

Lassen Sie nun das Ganze ein paar Mal laufen. Sie werden sehen, mit welcher Genauigkeit Sie Strecken berechnen können. Der Bereich geht dabei bis 99.59.59 Grad, was für geographische Breiten wenig sinnvoll ist, denn am Nordpol haben wir die maximale Breite von genau 90 Grad.

Doch niemand hindert Sie, das Programm so zu erweitern, daß Sie den gesamten Erdball umfassen können. Ein Vorschlag: Setzen Sie den Bereich für die geographischen Breiten von -90 bis +90 Grad und die Längen von -180 bis +180 Grad an. Dann ersparen Sie sich bei der Eingabe mit einer fortschrittlicheren »gradin«-Prozedur die Abfragen nach Nord/Süd bzw. Ost/West. Die negativen Breiten sind danach die südlichen, die negativen Längen die westlichen.

Diese Komfort-Version liefern wir Ihnen nicht. Sie sollten nämlich durch Selbst-Programmieren lernen, nicht nur durch Nachlesen. Das Werkzeug dazu haben wir Ihnen bereits komplett geboten.

6.5 Zusammenfassung (Funktionen)

- Funktionen liefern eindeutige Werte, also Zahlen, Strings oder ähnliches, nicht aber Felder oder Matrizen.

- Funktionen unterscheiden sich von Prozeduren dadurch, daß sie das Ergebnis immer unter dem Funktionsnamen beinhalten.

- Turbo kennt eigene, sogenannte vordeklarierte Funktionen, die nur mit einem festgelegten Namen (Funktionsbezeichner) aufgerufen werden. Die notwendigen Parameter werden in Klammern übergeben.

- Turbo läßt beliebige selbstdefinierte Funktionen zu, die mit einem beliebigen Namen bezeichnet und aufgerufen werden. Werden Pascal-eigene Bezeichnungen, sogenannte Standardbezeichner, verwendet, dann verlieren diese innerhalb des Blocks ihre Gültigkeit. Also, etwas Vorsicht ist geboten, solange man noch nicht alle Bezeichner, die Turbo bietet, im Kopf hat.

- Damit Sie nicht lange im Handbuch blättern müssen, finden Sie im Anhang B die Turbo-eigenen Funktionen einzeln aufgelistet.

Kapitel

7

Anwendungsbeispiele für Prozeduren und Funktionen

Mit dem Programm P13 sind wir in der Lage, die Entfernung zwischen zwei Punkten X und Y zu berechnen. Nun wollen wir auch noch den Kurs haben, den wir z.B. fliegen müßten, um vom Startort X zum Ziel Y zu gelangen. Wenn man diesen Kurs einer Karte entnimmt, dann spricht man vom sogenannten rechtweisenden Kurs, den wir kurz rwK nennen. Das ist also die Himmelsrichtung, in der das Ziel Y von X aus anzupeilen ist.

Um es gleich vorweg zu sagen: Eine Superformel zur exakten Kursberechnung wollen wir hier nicht aufstellen, uns genügt ein Verfahren zur Bestimmung eines brauchbaren Näherungswertes. Es ist gar nicht so kompliziert, wie man vermuten möchte, und hat natürlich die bereits vorhandenen geographischen Koordinaten zur Grundlage:

```
rwK=ArcTan(0.5*(Cos(nx)+Cos(ny))*dost/dnord)*180/Pi;
```

- dost=Längenunterschied, dnord=Breitenunterschied.
- nx,ny sind die geographische Breiten von Startort X bzw. Ziel Y.

Um noch einmal die Arbeitsweise einer Prozedur von der einer Funktion zu unterscheiden, berechnen wir diesmal den Kurs auf beide Arten. Die Ergebnisse müßten dann die gleichen sein.

Doch zunächst etwas Grundsätzliches zur Arbeit mit Winkeln und Winkelfunktionen. Wie Sie sehen, verwenden wir die Standardfunktion **ArcTan**, um den Kurs, der ja der Winkel zur Nordrichtung ist, zu berechnen. **ArcTan** erzeugt aus einem Tangenswert einen Winkel im Bogenmaß. Aber: Zum Tangenswert 1 gehören z.B. die Winkel 45, aber auch 225 Grad, zum Tangenswert -1 gehören -45 oder 135 oder 315 Grad.

Sie sehen, die Winkel erfahren keine eindeutige Zuordnung. Es gilt zwar: Zu jedem Winkel gehört genau ein Tangenswert. Aber es gilt nicht: Zu jedem Tangenswert gehört genau ein Winkel.

Aus diesem Grund gibt die Funktion **ArcTan** nur Winkel zwischen -90 und +90 Grad aus. Darüber hinausgehende Werte sind von uns selbst zu bestimmen. Das können wir auch; denn wenn man von West nach Ost fliegt, dann ist die Differenz der geographischen Längen zwischen Ziel und Start negativ. Von Ost nach West dagegen ist sie positiv. Bei den geographischen Breiten funktioniert das entsprechend, und das nützen wir natürlich aus.

7.1 Die Prozedur »RWKxy.P14« kann einen Kurs berechnen

Die Prozedur, die den Kurs liefern soll, nennen wir RWKxy:

```
(*A*) PROCEDURE rwKxy (nx,ny,ex,ey:Real;VAR rwK:Real);
      VAR
(*B*)   dost,dnord:Real;
      BEGIN
        dost:=ey-ex;dnord:=ny-nx;
(*C*)   IF (dnord=0) AND (dost<0) THEN rwK:=270;
        IF (dnord=0) AND (dost>0) THEN rwK:= 90;
        IF dnord <> 0 THEN
        BEGIN
(*D*)     rwK:=ArcTan(0.5*(Cos(nx)+Cos(ny))*dost/dnord)*180/Pi;
          rwK:=Round(rwK);
(*E*)     IF dnord<0 THEN rwK:=rwK+180;
          IF (dost<0) AND (dnord>0) THEN rwK:=rwK+360;
        END;
      END;
```

Erläuterungen:

(A) Die Koordinaten übernehmen wir als Festwertparameter. Sie müssen im Hauptprogramm im Bogenmaß übergeben werden.

(B) Mit den lokalen Variablen »dost« und »dnord« bilden wir die Differenzwerte der Längen bzw. der Breiten.

(C) Sollte die Breitendifferenz Null sein, dann handelt es sich auf jeden Fall entweder um einen östlichen oder einen westlichen Kurs. Ist die Längendifferenz zugleich (**AND**) positiv, dann liegt eindeutig ein Ostkurs an, also ist rwK=90 Grad. Bei Westkursen (270 Grad) gilt Entsprechendes.

(D) Wenn die Sonderbedingungen Ost oder West nicht zutreffen, rechnen wir den Kurswinkel mit der Formel aus und runden ihn auf eine ganze Zahl.

Damit erhalten wir Gradzahlen je nach Lage der Anfangs- und Endpunkte zwischen -90 und +90 Grad. Mehr läßt **ArcTan** nicht zu.

(E) Wieder können wir mit Hilfe der Koordinatendifferenzen entscheiden, ob die berechnete Kurszahl noch einer Korrektur bedarf oder nicht: Wenn der Unterschied zwischen den Breiten nämlich negativ ist, handelt es sich um eine Richtung, die eine Südkomponente haben muß. Deshalb addieren wir 180 Grad.

Sollte **ArcTan** eine negative Zahl ausgeben, ist sie um 360 Grad zu erhöhen, aber nur, wenn es sich um nördliche Kurse handelt.

Sollten Sie die mathematischen Zusammenhänge gelangweilt haben, dann können Sie wieder aufwachen: Die Variable »rwK« enthält jedenfalls jetzt einen Wert zwischen 0 und 360. Und dieser Wert wird von dem letzten Parameter übernommen, der im Hauptprogramm steht, das wir nun selbstverständlich um folgende Zeile erweitern müssen:

```
rwKxy(breite1D, breite2D, laenge1D, laenge2D, kurs_pd);
```

»kurs_pd« ist eine neue Variable, die wir im globalen Deklarierungsteil als Realtyp definiert unterbringen müssen. Sie enthält nach Beendigung der Prozedur »rwKxy« den richtigen Kurs als Dezimalzahl. Wir benötigen sie, damit wir ihn auch ausgeben können. Deshalb am Schluß des Hauptteils die Zeile:

```
WriteLn('Kurs aus der Prozedur: ',kurs_pd:4:0);
```

7.2 Die Funktion »kursxy.F15« liefert das gleiche Ergebnis

Wie man das gleiche Ergebnis mit Hilfe einer Turbo-Funktion erzeugt, sehen Sie im nächsten Beispiel:

```
(*A*) FUNCTION kursxy (nx,ny,ex,ey:Real):Real;
      VAR
(*B*)   dost,dnord,rwK:Real;
      BEGIN
(*C*)   dost:=ey-ex;dnord:=ny-nx;
        IF (dnord=0) AND (dost<0) THEN rwK:=270;
        IF (dnord=0) AND (dost>0) THEN rwK:=90;
        IF dnord <> 0 THEN
        BEGIN
        rwK:=ArcTan(0.5*(Cos(nx)+Cos(ny))*dost/dnord)*180/Pi;
          rwK:=Round(rwK);
          IF dnord <0 THEN rwK:=rwK+180;
          IF (dost < 0) AND (dnord >0) THEN rwK:=rwK+360;
        END;
(*D*)   kursxy:=rwK;
      END;
```

Erläuterungen:

(A) Wir haben einen anderen Namen für die Funktion gewählt, weil wir beide in dasselbe Hauptprogramm eingliedern wollen.

Der wichtigste Unterschied bei den Parametern: Es gibt keine Extra-Variable für das Ergebnis, »kursxy« wird es aufnehmen.

(B) Wir führen die lokale Variable »rwK« ein, dann können wir den Rechenteil genau so übernehmen, wie er in der Prozedur »rwKxy« steht.

(C) Die Berechnung des Kurswerts erfolgt genau wie bei der vorhin besprochenen Prozedur.

(D) Nach dem fertig erstellten Ergebnis weisen wir dieses dem Funktionsbezeichner »kursxy« zu.

Im Hauptprogramm nehmen wir die Zeile auf, die den Funktionsaufruf beinhaltet. Sie muß nun im Gegensatz zu der Prozedur den Inhalt von »kursxy«

irgendwie verarbeiten. Falsch wäre es, zu versuchen, die Funktion allein als Anweisung hinzustellen.

So ist es richtig:

```
kurs_fn:=kursxy (breite1D,breite2D,laenge1D,laenge2D);
```

Die neue Variable »kurs_fn«, der wir den Kurswert zuweisen, den die Funktion berechnet, nehmen wir wieder zusätzlich in unser Hauptprogramm auf (Typ Real).

Und nun lassen wir uns in einer wirklich letzten Anweisung diesen Wert auf dem Schirm ausgeben:

```
WriteLn('Kurs aus der Funktion: ', kurs_fn:4:0);
```

Auch ohne den Umweg über »kurs_fn« ist eine Ausgabe – allerdings unübersichtlicher – möglich mit:

```
WriteLn(kursxy(breite1D,...,laenge2D));
```

Markieren Sie nun die beiden Blöcke, und speichern Sie sie unter den Namen »rwKxy.P14« bzw. »rwKxy.F15« getrennt auf Diskette ab. Wir beschränken uns später auf eine einzige Version, denn wenn Sie mal einen Testlauf durchgeführt haben, werden Sie sich am identischen Ergebnis freuen, falls Sie keine Fehler eingebaut haben.

Das Einbinden der beiden Routinen sollte Ihnen keine Schwierigkeiten bereitet haben. Sie arbeiten doch sicher schon mit **Include**-Dateien und haben folgende Anweisungen verwendet:

```
{$I rwKxy.P14 }
{$I rwKxy.F15 }
```

Weil es zu diesem Buch keine Begleitdiskette gibt, drucken wir am Ende dieses Kapitels das Listing einer funktionstüchtigen Version dieses Programms ab. Nennen wir es »P13_RWK.kpl« Sie können es als Ganzes sichern oder aber die Version mit der Include-Datei realisieren, wenn Sie die Funktion »rwKxy.F15« bereits auf Diskette bereit haben.

Geben Sie beim Testen des Programms auch mal Orte mit ungewöhnlichen Koordinaten ein. Sie werden bei den Ergebnissen dabei auf ein paar Eigenheiten stoßen, die wir hier

aber nicht abhandeln wollen. Das Navigationsbeispiel war schließlich nur Mittel zum Zweck, sprich Turbo. Falls Sie aber die Absicht haben, mal einen Flugsimulator oder ähnliches zu erstellen, werden Sie schnell wieder auf diese Problematik stoßen.

Anmerkung: Im folgenden Listing des Programms P13_RWK.PAS wird eine verbesserte Eingabeprozedur eingesetzt, die Sie unter dem Namen »gradin« finden.

```
(*************************************************************)
(*          TURBO PASCAL auf dem Schneider CPC 6128          *)
(*               Strecken- und Kursberechnung                *)
(*                 Markt & Technik MT 90455                  *)
(*                   (C) W. Kassera 1987                     *)
(*************************************************************)

     PROGRAM P13;   (* Streckenberechnung *)
     TYPE
       grad=String[8];
     VAR
       breite1,breite2,laenge1,laenge2:grad;
       breite1d,breite2d,laenge1d,laenge2d:Real;
       distanz:Real;
       kurs_pd,kurs_fn:Real;

     PROCEDURE gradin (VAR grin:grad);
     VAR  (* Deklarierung der lokalen Variablen *)
       z:Integer;
       ziffern:Set Of Char;
       ch:Char;
     BEGIN
       ziffern:=['0'..'9'];
       grin:='';z:=0;
       REPEAT
         Read (KBD,ch);
         IF ch IN ziffern THEN
         BEGIN
           IF NOT  (((z=2) OR (z=4)) and (ch > '5')) THEN
           BEGIN
             z:=z+1;
             Write(ch);
             grin:=grin+ch;
             IF  (z=2) OR (z=4) THEN
             BEGIN
               Write('.');
               grin:=grin+'.';
             END;
```

```
      END;
    END;
    IF (ch=#127) AND (Length(grin) > 0 ) THEN
    BEGIN
      Write(^H,' ',^H); grin:=Copy(grin,1,Length(grin)-1);
      IF (z=2) OR (z=4) THEN
      BEGIN
        Write(^H,' ',^H); grin:=Copy(grin,1,Length(grin)-1);
      END;
      z:=z-1;
    END;
  UNTIL  (ch=^M)  {OR  (Length  (grin)=8)};
END;

PROCEDURE grad_dez_rad  (koo:grad;VAR kood:Real);
(* gradrad.p10 *)
VAR
  grad,min,sec,y:Integer;
BEGIN
  kood:=0;
  Val(Copy(koo,1,2),grad,y);
  Val(Copy(koo,4,2),min,y);
  Val(Copy(koo,7,2),sec,y);
  kood:=grad + min/60 + sec/3600;
  kood:=kood*Pi/180;
END;

(* Streckenberechnung Funktion P12.FKN *)
FUNCTION arccos(cosstr:Real):Real;
BEGIN
  arccos:=ArcTan(Sqrt(Abs(1-cosstr*cosstr))/cosstr);
END;
FUNCTION strecke (b1,l1,b2,l2:Real):Real;
VAR
  cosstr: Real;
CONST
  erdradius:Real = 6378.245;
BEGIN
  cosstr:=Sin(b1)*Sin(b2)+Cos(b1)*Cos(b2)*Cos(Abs(l1-l2));
  strecke:=erdradius*arccos(cosstr);
END;

FUNCTION kursxy  (nx,ny,ex,ey:Real):Real;
VAR
  dost,dnord,rwk:Real;
BEGIN
  dost:=ey-ex;dnord:ny-nx;
  IF (dnord=0) AND (dost<0) THEN rwk:=270;
```

```
  IF  (dnord=0)  AND  (dost>0)  THEN  rwk:=90;
  IF  dnord <> 0 THEN
  BEGIN
    rwk:=ArcTan(0.5*(Cos(nx)+Cos(ny))+dost/dnord)*180/Pi;
    rwk:=Round(rwk);
    IF  dnord < 0  THEN  rwk:=rwk+180;
    IF  (dost < 0) AND  (dnord > 0)  THEN  rwk:=rwk+360;
  END;
  kursxy:=rwk;
END;

BEGIN
  ClrScr;
  WriteLn('Streckenberechnung aus geographischen Koordinaten');
  Write(^J^J^J^M,'Ort 1, Breite: ');    GradIn(breite1);
  grad_dez_rad(breite1,breite1d);
  Write('   Laenge: ');
  gradin(laenge1);
  grad_dez_rad(laenge1,laenge1d);
  Write(^J^M,'Ort 2, Breite: ');
  gradin(breite2);
  grad_dez_rad(breite2,breite2d);
  Write('   Laenge: ');
  gradin(laenge2);
  grad_dez_rad(laenge2,laenge2d);
  distanz:=strecke(breite1d,laenge1d,breite2d,laenge2d);
  WriteLn(^J^J^M,'Strecke in km: ',distanz:9:3);
  kurs_fn:=kursxy(breite1d,breite2d,laenge1d,laenge2d);
  WriteLn('Kurs aus der Funktion: ',kurs_fn:4:0,' Grad');
END.
```

Kapitel

8

Funktion ruft Funktion

8.1 Vorsicht – Rekursion!

Wenn eine Funktion innerhalb ihres Anweisungsteils sich selbst aufruft, dann spricht man von Rekursion.

Dabei können für den Programmablauf recht unangenehme Folgen auftreten, wenn man keine Sicherheitsmaßnahmen trifft. Denn um ein Unterprogramm bei der Ausführung zu verwalten, werden auf dem sogenannten Stack, einem Bereich im Arbeitsspeicher des Computers, Zwischenergebnisse, Rücksprungadressen usw. abgelegt. Dieser Stack wächst mit zunehmender Datenmenge in Richtung der kleineren Speicheradressen dem sogenannten Heap entgegen, der wiederum besondere Variablen stapelartig gespeichert hat.

Nun kann man sich vorstellen, was passiert, wenn die Funktion sich ständig selbst aufruft: Der Stack und der Heap wachsen aufeinander zu und würden sich ineinander verkeilen, was aber unser recht intelligentes Turbo-System verhindert, weil es bei jedem Aufruf prüft, ob noch genügend Zwischenraum vorhanden ist. Ist dies nicht der Fall, bricht Turbo ab, bevor Schlimmeres passiert, und meldet einen Fehler: »Heap/Stack-Collision«.

Manchmal jedoch findet die Überschneidung vor dem Eingreifen von Turbo statt, und der Rechner »verschwindet im Wald«. Jetzt hilft nur noch ein Reset mit den Tasten **<CTRL/SHIFT/ESC>**. Hoffentlich war das Programm vorher abgespeichert, sonst ärgert man sich über die eigene Nachlässigkeit.

Unter CP/M kennt Turbo drei Variablen, mit denen man ständig den letzten Stand des Heap und des Stack kontrollieren kann. Es sind dies **HeapPtr**, **RecurPtr** und **StackPtr**, sogenannte Zeiger, die Turbo ständig mitführt. Mit ihrer Hilfe lassen sich auch Rekursionen überwachen. Dazu betrachten wir die interne Verwaltung mal an einer Stelle etwas genauer:

CP/M-80 unterhält speziell für Rekursionen einen eigenen Stack, der anfangs 1Kbyte unter dem Stack liegt, der vom Prozessor beansprucht wird (CPU-Stack).

Bei rekursiven Aufrufen wandert demnach der Rekursionsstack dem Heap entgegen, gleichzeitig füllt sich aber auch der CPU-Stack in Richtung des Rekursionsstacks auf (siehe Bild 8-1). Wenn man das weiß, wundert man sich nicht mehr über gelegentliches Aussteigen des Rechners.

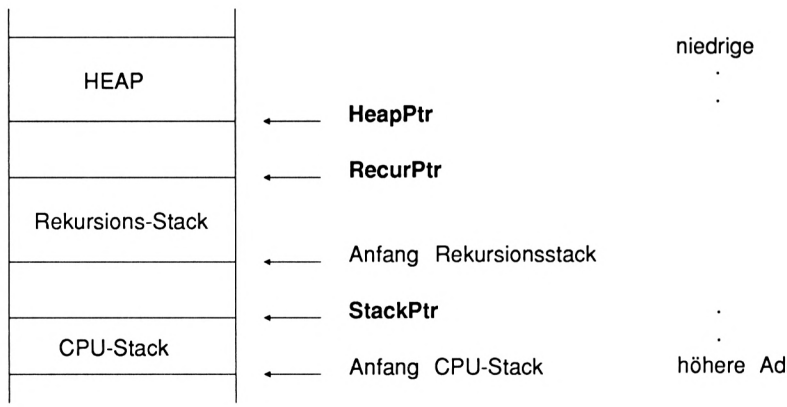


Bild 8-1: *Heap, Rekursions- und CPU-Stack*

Alle diese Eigen- bzw. Unarten treten erst zutage, wenn das Programm compiliert und zum Start freigegeben wurde.

Um einen Code zu erzeugen, der rekursive Aufrufe zuläßt, muß unter CP/M der Compilerbefehl **{$A-}** vor den Programmtext gesetzt werden, bevor die eigentliche Rekursionroutine anfängt.

Kommt der Rechner jedoch wieder rechtzeitig von einer rekursiven Funktion oder Prozedur zurück, wird der gesamte Speicherplatz, den dieses Unterprogramm benötigt hatte, wieder freigegeben. Die Zeiger werden wieder zurückgesetzt, und Turbo tut so, als ob es nicht wüßte, daß das Programm soeben einem fast immer tödlichen Absturz entgangen ist.

Doch so gründlich brauchen wir das für unsere praktische Programmierarbeit gar nicht zu wissen. Es genügt zunächst, wenn Ihnen bekannt ist, daß es rekursive Aufrufe gibt und daß man sie tunlichst nicht sich selbst überlassen, sondern auf ihrem Weg ins Unendliche rechtzeitig abfangen sollte.

Doch damit sei der Theorie Genüge getan. Es wird höchste Zeit, daß wir unser Wissen an einem Beispiel erproben. Wir empfehlen Ihnen dringend, das nächste Programm intensiv zu testen.

8.2 Quadratwurzel rekursiv berechnet Programm P16: Rekursion, IF..THEN, Ln, Exp

Die meisten Schüler kommen in ihrem Mathematikunterricht irgendwann einmal zu den irrationalen Zahlen, zu denen z.B. auch die Quadratwurzel aus 1000 gehört. Dabei lernen sie ein Verfahren kennen, das durch wiederholte Anwendung einen immer genauer werdenden Wert für die Zahl liefert, die mit sich selbst multipliziert 1000 ergibt. Will man diesen Wert nämlich mit einer Dezimalzahl darstellen, dann kommt man zu keinem Ende. Die Nachkommastellen laufen in unregelmäßiger Folge ins Unendliche. Genau das ist eine typische Eigenschaft einer irrationalen Zahl.

Es ist also nur möglich, solche Zahlen mit einer bestimmten Genauigkeit, sagen wir auf sieben oder acht Kommastellen, zu berechnen. Die Genauigkeit hängt immer davon ab, wie man die Zahl weiter verarbeiten möchte.

Schauen wir uns einen solchen Algorithmus (das ist ein Verfahren in endlich vielen Schritten) an.

Unser Ziel: Wir suchen die Quadratwurzel aus 1000. Damit ist 1000 unser Radikand R.

1. Schritt:
Annahme eines willkürlichen Anfangswerts Xneu für das gesuchte Ergebnis (z.B. Xneu=1).

2. Schritt:
Da mit diesem Ausgangswert ein genauerer Wert Xneu berechnet werden soll, weisen wir zur Unterscheidung Xneu der Variablen Xalt zu (z.B. Xalt=1).

3. Schritt:
Dividieren wir den Radikanden R durch Xalt, dann erhalten wir einen weiteren Wert Xzu und können damit den Radikanden R in die zwei Faktoren Xalt und Xzu zerlegen, die anfangs sicher nicht gleichwertig sind, wenn wir nicht gerade die Wurzel aus 1 suchen.

Dabei gilt: Xzu ist zu klein, wenn Xalt größer war als das gesuchte Ergebnis, und Xzu ist zu groß, wenn Xalt kleiner war als das gesuchte Ergebnis.
(In unserem Beispiel ergibt sich anfangs: Xalt=1 und Xzu=1000.)

4. Schritt:
Der Mittelwert aus Xalt und Xzu liefert einen besseren Näherungswert Xneu. (In unserem Beispiel anfangs: Xneu=0.5*(1+1000), also Xneu=500.5.)

5. Schritt:
Überprüfung, ob Xneu schon als Ergebnis akzeptiert werden kann. Dabei bieten sich zwei Möglichkeiten an:

a) gilt Xalt=Xzu ?
b) gilt Xneu*Xneu=R ?

(In unserem Beispiel anfangs: 500.5*500.5=1000 ?)

Genügt der gefundene Wert Xneu unseren Anforderungen, dann können wir zu Schritt 6 springen.

Ist die gewünschte Genauigkeit noch nicht erreicht, dann nehmen wir Xneu als Ausgangswert für die Berechnung eines besseren Werts. Schritt 2 leitet dies ein. Also Sprung dorthin.

6. Schritt:
Xneu als Ergebnis sichern, ausgeben, oder ähnliches.

Im Ablaufplan sieht dieses sogenannte Heronsche Verfahren so aus:

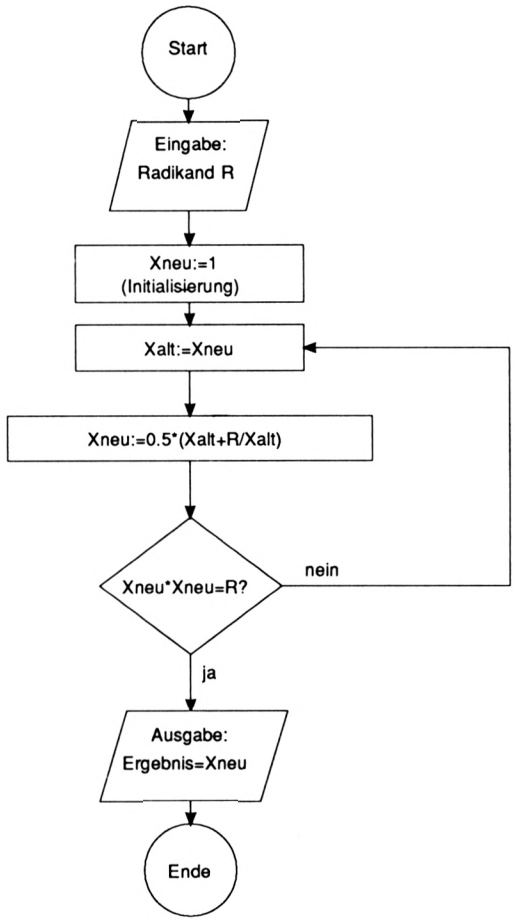


Bild 8-2: *Ablaufplan zum Heronschen Verfahren*

Mit neun Schleifendurchgängen erzielt man bereits für die Wurzel aus 1000 eine Genauigkeit auf 2 Stellen nach dem Komma. Folgende Werte werden dabei errechnet:

Xalt	Xzu
1.00	1000.00
500.50	2.00
251.25	3.98
127.62	7.84
67.73	14.77
41.25	24.25
32.75	30.54
31.64	61.60
31.62	31.62

Zur Kontrolle geben wir die Werte aus, die von der Funktion ermittelt wurden.

Die Formel, die man aufstellen kann, und die man immer wieder verwendet, lautet also:

```
Xneu=0.5*(Xalt+zahl/Xalt),
```

wenn von »zahl« die Quadratwurzel bestimmt werden soll und der neue Näherungswert »Xneu« aus dem vorhergehenden »Xalt« berechnet wird.

Dieses ständig wiederholende Aufrufen der gleichen Operation nennt man Iteration, also ein schrittweises Verfahren.

Das kann unser Rechner viel schneller. Schauen wir uns das zugehörige Pascal-Programm einmal an.

```
        PROGRAM P16; (* Rekursion *)
        VAR
(*A*)     zahl:Real;
          stelle,rptranfang:Integer;
(*B*)   {$A-}

(*C*)   FUNCTION ergebnis (zahl,Xalt:Real):Real;
        VAR
          Xneu:Real;
        BEGIN
          WriteLn(Xalt:10:3,zahl/Xalt:10:3);
(*D*)     Xneu:=0.5*(Xalt+zahl/Xalt);
(*E1*)    IF RecurPtr-HeapPtr<10 THEN
          BEGIN
            WriteLn(Xneu:25:stelle,'  R/H-Crash drohte');Halt;END;
            IF StackPtr-rptranfang<10 THEN
```

```
            BEGIN
            WriteLn(Xneu:25:stelle,'  R/S-Crash drohte');Halt;END;
              IF Xneu*Xneu=zahl THEN ergebnis:=Xneu *)
(*E2*)    IF Xneu*Xneu-zahl < 1/Exp(stelle*Ln(10))
          THEN ergebnis:=Xneu
(*F*)     ELSE ergebnis:=ergebnis(zahl,Xneu);
        END;

        BEGIN (* Hauptprogramm *)
          rptranfang:=RecurPtr;
          ClrScr;
          WriteLn('Quadratwurzelberechnung mit Rekursion'^J^J^M);
          Write('Radikand: ');
(*G*)     ReadLn(zahl);
          Write('Genauigkeit auf wieviele Nachkommastellen: ');
          ReadLn(stelle);
(*H*)     Write(ergebnis(zahl,1):1:stelle);
          WriteLn(' = Wurzel aus ',zahl:1:10);
        END.
```

Erläuterungen:

(A) »zahl« ist der Wert, aus dem die Quadratwurzel gezogen werden soll. »stelle« nimmt die Anzahl der Nachkommastellen des Ergebnisses auf. In »rptranfang« halten wir den usprünglichen Stand des Rekursionszeigers fest.

(B) Da wir rekursive Aufrufe beabsichtigen, teilen wir dies dem Compiler mit **{$A-}** mit.

(C) Die Funktion »ergebnis« benötigt die Übernahmeparameter »zahl" und den bereits vorhandenen Näherungswert »Xalt«.

Die Variable »Xneu« dient zum Berechnen einer neuen Näherung aus den übernommenen Parametern.

(D) Das ist die oben besprochene Iterationsformel in Pascal-Schreibweise.

(E) Zum Ausprobieren bieten wir Ihnen zwei Möglichkeiten an, von denen hier im Programm nur die zweite aktiv ist. Die andere wurde durch Kommentarklammern stillgelegt.

(E1) Diese Abbruchbedingung hört erst dann mit der Rekursion auf, wenn das

exakte Ergebnis erzielt ist. Das kann eigentlich nie der Fall sein, es sei denn, der Rechner erzeugt mit seinem Binärcode bei irrationalen Zahlen solche Werte, daß das Ergebnis richtig gerundet wird. (Tatsächlich ist das für »zahl«:=2 der Fall.) Oder aber die »zahl« war eine Quadratzahl.

Zur Absicherung gegen einen Absturz berechnen wir die Differenz zwischen den gefährdeten Speicherbereichen. Dazu wird mit »rptranfang« die Anfangsadresse des Rekursionszeigers zu Beginn des Hauptprogramms festgehalten. Drohende Kollisionen werden gemeldet, wenn sich die Zeiger bedenklich nahegekommen sind, und die Rekursion wird mit dem Turbo-Befehl **Halt** abgebrochen. Der bis zu diesem Zeitpunkt ermittelte Näherungswert kann selbstverständlich ausgegeben werden.

(E2) Mit dieser Abbruchbedingung verläßt man die Funktion dann, wenn die Genauigkeit auf eine vorbestimmte Nachkommastelle erreicht ist. Auch das hat seine Grenzen in der beschränkten Anzahl Stellen der Realzahlarithmetik unseres Rechners.

Was die Funktionen **Exp** und **Ln** hier zu suchen haben, erklären wir ausführlich im nächsten Abschnitt.

(F) Ist die Abbruchbedingung nicht erfüllt, berechnen wir mit Hilfe des eben erhaltenen »ergebnis« ein neues »ergebnis«.

Hier liegt das eigentliche Herzstück der Rekursion: Dem »ergebnis« wird wieder ein Ergebnis zugewiesen, das durch die Funktion »ergebnis« erst berechnet werden muß. Klingt doch ganz lustig, oder?

(G) Im Hauptprogramm lassen wir uns nach der gewünschten »zahl« fragen und nach der Anzahl der auszurechnenden Nachkommastellen.

Wer besonders pingelig ist, der läßt sich bei der Schirmausgabe eine Stelle weniger zeigen als berechnet wurde, denn die ist noch gesichert.

(H) Wie man sieht, kann das »ergebnis« sofort weiterverarbeitet werden. Hier in der formatierten **WriteLn**-Anweisung.

Probieren Sie nun das Programm P16 mit den zwei unterschiedlichen Abbruchbedingungen aus. Speichern Sie es aber unbedingt vorher ab. Wir sind beim Ausprobieren nämlich ebenfalls mehrmals im Chaos gelandet.

Besonders die Version (E1) ist gefährlich. Sie führt in der Regel zum Ausstieg des Programms. Geben Sie mal die Zahlen 100, 2, 0.25 und 1000 zum Berechnen ein!

Beachten Sie bitte, daß Turbo zwar eine Überschneidung von Heap und Stack kontrolliert, aber nicht auf eine Kollision von CPU- und Rekursionsstack reagiert. Legen Sie in (E1) einmal die R/S-Prüfung durch Kommentarklammern still, und beobachten Sie, was passiert.

Zur Form:

- Die Anweisung **Halt** beendet einen Programmteil an dieser Stelle. Jedoch verlangt der Compiler trotzdem den Abschluß eines Anweisungsblocks mit **End**.

Nun aber noch zu den bisher nicht besprochenen Funktionen **Exp** und **Ln**, die – da sie im Programm nicht deklariert wurden – Standardfunktionen sein müssen.

8.3 Anwendung der Standardfunktionen Ln und Exp

In manchen Pascal-Büchern werden diese Funktionen recht stiefmütterlich behandelt, und manchmal hat man den Eindruck, die Verfasser wüßten selbst nichts Rechtes mit ihnen anzufangen.

Ein paar kurze Erklärungen:

Ln bildet den sogenannten natürlichen Logarithmus aus einer beliebigen Zahl. Genauer gesagt wird mit LN(10) die Frage beantwortet, welcher Exponent zur Basis e (e ist die sogenannte Eulersche Zahl: 2.718281828....) notwendig ist, damit der Potenzwert 10 ergibt. In unserem Fall wäre das etwa 2.302585093...

Mit anderen Worten: *e hoch 2.30258... ist gleich 10,*
oder: *der Logarithmus von 10 zur Basis e ist gleich 2.302585....*

Man sieht, hier wird mit irrationalen Zahlen gearbeitet. Warum man ausgerechnet die Zahl e als Basis für diese Operationen verwendet, ist hier nicht mit wenigen Worten erklärbar. Wenn Sie das genau wissen wollen, schlagen Sie am besten in einem Mathematikbuch nach.

Was soll das alles ?

Nun, mit Hilfe von Logarithmen läßt sich eine höhere Rechenart immer auf die nächst niedrigere zurückführen:

- das Potenzieren auf das Multiplizieren,
- das Multiplizieren auf das Addieren,
- das Wurzelziehen auf das Dividieren und
- das Dividieren auf das Subtrahieren.

Ein einfaches Beispiel, das wir leicht im Kopf nachprüfen können: Berechnen wir mal 10 hoch 6. Klarer Fall, Ergebnis: 1 Million.

Mit der Logarithmusfunktion berechnen wir zunächst den Logarithmus des Potenzwerts: das macht 6 mal LN(10). Was dabei herauskommt, braucht uns eigentlich überhaupt nicht zu interessieren (es macht ca. 13.815510...).

Wichtig wäre jetzt nur, daß wir wieder zurück auf unsere übliche Rechenebene finden. Und genau das macht die Funktion **Exp**. Sie bildet wieder die Umkehrung zum Logarithmus. Schreiben wir nun x:=**Exp**(6*Ln(10)), dann erhalten wir für x tatsächlich unsere 1000000.

Warum wir das so ausführlich besprechen? Weil selbst Turbo keine vordeklarierten Funktionen für solche Arbeiten hat. Mit Zehnerpotenzen rechnet es sich ja noch einfach. Aber was machen Sie, wenn Sie mal die dritte Wurzel aus 48.6 ziehen müssen?

Aha, Sie wenden ein Iterationsverfahren an und programmieren eine rekursive Funktion, die so lange aufgerufen wird, bis ein brauchbares Ergebnis zustande kommt. Sie könnten es aber auch einfacher haben, wenn Sie die bereits vorhandenen Standardfunktionen **Ln** und **Exp** verwenden. Wir verraten Ihnen sogar das Ergebnis:

x:=**Exp**(1/3*Ln(48.6)), es liefert den Wert ca.3.6493211...

Prüfen Sie das ruhig mit Ihrem Taschenrechner, aber vor allem mit einer kleinen Pascal-Routine nach.

Zusammenfassung:

Die Logarithmusfunktion **Ln** und ihre Umkehrfunktion **Exp** haben das Rechnen mit Potenzen zur Grundlage. Die Basis (Grundzahl) für diese Funktionen ist die Zahl e.

Mit **Ln** und **Exp** lassen sich höhere Rechenarten wie Potenzieren und Wurzelziehen in beliebigen Schwierigkeitsgraden bewältigen.

Damit kennen wir nun alle arithmetischen Funktionen, die Turbo zu bieten hat. Es sind dies:

- **Abs**(zahl)
- **ArcTan**(zahl)
- **Cos**(zahl)
- **Exp**(zahl)
- **Frac**(dezimalzahl)
- **Int**(dezimalzahl)
- **Ln**(zahl)
- **Sin**(zahl)

Der Parameter »dezimalzahl« soll hier nur andeuten, daß es sich um Operationen mit Nachkommastellen handelt.

Übrigens: Ist Ihnen jetzt klar, was die Zeile bei (E2) im Programm P16 bedeutet?

Dort heißt es: `(*E2*) IF Xneu*Xneu-zahl < 1/Exp(stelle*Ln(10))...`

In Worten: Wenn der Unterschied zwischen dem quadrierten Ergebnis und der gegebenen Zahl kleiner ist als die vorgegebene Dezimalstelle, dann sind wir mit dem Ergebnis zufrieden.

Statt dem Ausdruck

```
1/Exp(stelle*Ln(10))
```

können Sie auch schreiben:

```
Exp(-stelle*Ln(10))
```

das ist ebenfalls der Kehrwert.

Ist z.B. »stelle«=5, dann beträgt der Wert dieses Terms ein Hunderttausendstel. Und mit dieser erreichten Genauigkeit wird der Iterationsprozeß abgebrochen.

Kapitel

9

Übersichtliche Daten mit strukturierten Typen

Daten, die in sich geordnet sind, nennt man strukturierte Daten. Dazu gehören die sogenannten **Arrays** (auch Matrizen oder Felder genannt), die sogenannten **Records** (Datensätze mit Untergliederungen) und natürlich die Daten, die auf einem Speichermedium verwaltet werden, z.B. also Diskettendateien. Im letzteren Fall leuchtet es sofort ein, daß da Ordnung herrschen muß und nicht jedes Datum angeordnet sein kann, wie es der Zufall gerade will.

Aber auch bereits in einem Programm ist es sehr hilfreich, wenn man sich die benötigten Daten in sinnvollen Strukturen aufbereitet. Man spart sich und dem Rechner Zeit und erhält einen besseren Überblick, was da so alles hin- und hergeschoben wird.

Eins geben wir zu: Es macht zunächst einmal etwas Mühe, die einzelnen Möglichkeiten, die Turbo in dieser Richtung zu bieten hat, aufzunehmen und zu durchdenken. Aus eigener Erfahrung wissen wir, wie zäh gerade auf dem Gebiet der strukturierten Daten die Fortschritte sind, wenn man sich alle Einzelheiten selbst erarbeiten muß. Versuchen wir daher im folgenden, so gründlich wie notwendig und so behutsam wie möglich die erforderliche Denkweise aufzubauen.

Dabei bleiben wir – soweit dies sinnvoll ist – an unserem Beispiel, das uns bisher schon erlaubt, Namen von Orten und ihre Koordinaten aufzunehmen, zu verarbeiten und wieder auszugeben. Wir haben schon mehrere solche Input/Output-Mechanismen (kurz I/O) kennengelernt. Tun wir dies nun auch mit strukturierten Datentypen!

9.1 Eindimensionale Felder Programm P17: Array-Demonstration

Ein eindimensionales (lineares) Feld oder **Array** hat in allen höheren Programmiersprachen einen Namen, unter dem lauter gleichartige Daten in abzählbarer Ordnung abgelegt werden können. Wem das zu kompliziert klingt, der kann sich das an einem Beispiel veranschaulichen:

Nehmen wir an, wir nehmen eine Liste von 10 Ortsnamen auf, die wir später wieder weiterverarbeiten wollen. Dann können wir so vorgehen, daß wir jedem dieser 10 Daten, die in diesem Fall vom Typ her Strings sind, jeweils einen Variablennamen zuordnen.

Wir müssen dann eine Liste führen von »name1«, »name2«... bis »name10«. Um direkt auf »name6« zugreifen zu können, um vielleicht die zugehörigen geographischen Koordinaten abzurufen, können wir nicht einfach über die Nummer 6 die entsprechenden Werte abfragen. Denn um festzustellen, ob es sich bei einem beliebigen Ortsnamen auch wirklich um Ort Nr. 6 handelt, müssen wir vielleicht erst den Variablennamen untersuchen, ober er eine »6« enthält.

Jetzt könnten wir eine einzige Variable brauchen, deren Komponenten sich nur durch ihre Nummer (den sogenannten Index) unterscheiden. Auch Pascal kennt solche indizierten Variablen. Es sind die eindimensionalen **Arrays**.

Wie legt man solche Arrays nun an?

Zunächst muß auf jeden Fall einmal der Typ der Komponenten definiert werden, die in diesem **Array** untergebracht werden sollen. Das geschieht am besten im Teil **TYPE**. Noch einmal: Wichtig ist, daß alle diese Komponenten vom gleichen Typ sind, sonst hätte ja ein **Array** auch keinen Sinn.

In unserem Beispiel sollen Ortsnamen eine Länge von höchstens 20 Zeichen haben, also wählen wir als Typ **String**[20] und bezeichnen diesen Typ mit »name«. Soweit ist alles bekannt.

Das **Array** ist ein besonderer Variablentyp und muß ebenfalls deklariert werden, also auch im Teil **TYPE**, aber natürlich erst, nachdem die Komponente »name« eingeführt worden ist, also dahinter. Benötigt wird auch noch die Ordnung, unter der die Komponenten im **Array** abgelegt und aufgerufen werden sollen.

Meistens verwendet man dazu Ordnungszahlen und beginnt mit 1 oder computergerecht mit 0 zu zählen. Das muß aber nicht sein, Turbo schenkt hier grenzenlose Freiheit. Sie können bei Belieben auch eine Ordnung mit (rot, blau, gelb,...), also mit Farben oder irgendwelchen mehr oder weniger sinnvollen Bezeichnungen aufbauen.

Damit wird aber auch gleich die maximale Anzahl der Komponenten dieses **Arrays** festgelegt. Mit Zahlen tut man sich da leicht:

[1..100] heißt eben »von 1 bis 100«. Die zwei Punkte sind hier möglich, weil die Ordnung bereits vorgegeben ist. Auch Buchstaben haben durch ihre Codierung (ASCII) ihre feste Ordnung. Ohne weiteres ist also folgende Dimensionierung möglich: [a..h,A..C]. Damit sind 11 Elemente vorgesehen.

Legen wir nun für den Typ unseres **Arrays** die Bezeichnung »namenfeld« fest, dann kann der Typ des **Arrays** so angegeben werden:

```
namenfeld=Array[1..100] Of String[20];
```

Schließlich muß noch der Variablenname festgelegt werden, unter dem alle Orte laufen sollen. Das geschieht im **VAR**-Teil und kann so aussehen:

```
name: namenfeld;
```

Wem das zu umständlich erscheint, der kann sein **Array** auch auf einen Schlag unter **VAR** deklarieren:

```
name: Array[1..100] Of String[20];
```

Man nimmt sich aber damit die Möglichkeit, beim Auftreten gleichartiger **Arrays** die Typen mehrmals verwenden zu können.

Die einzelnen Komponenten des **Arrays** »name« werden aber in beiden Fällen mit dem **Array**-Namen und dem Ordnungsbezeichner aufgerufen. Dieser muß natürlich in der Dimensionierung enthalten sein. In unserem Beispiel laufen die Komponenten von

`name[1], name[2],...` bis `name[100].`

Genau in dieser Weise lassen sich auch die geographischen Breiten- und Längengrade erfassen. Der Vorteil: Durch ihre innere Ordnung sind alle drei Angaben eines Punktes, also Name, Breite und Länge, eindeutig einander zugeordnet.

Nun das Beispielprogramm P17:

```
      PROGRAM P17; (* Array eindimensional *)
      TYPE
        grad=String[8];
        angaben=String[38];
      VAR
(*A*)   breite,laenge:Array [1..100] Of grad;
        name:Array [1..100] Of String[20];
        navpunkt:Array [1..100] Of angaben;
        n:Integer;
      CONST
(*B*)   tz:Char ='/';
      {$I GRADIN.PD}
      BEGIN
        ClrScr;WriteLn('Erfassung von 3 Navigationspunkten'^J^J^M);
(*C*)   FOR n:=1 TO 3 DO (*Dreieckstrecke*)
        BEGIN
          BufLen:=20;
          Write('Ortsbezeichnung Nr.',n,': '); Read(name[n]);
          Write('  Breite: ');        gradin(breite[n]);
          Write('  Laenge: ');        gradin(laenge[n]);
          WriteLn;
(*D*)     navpunkt[n]:=name[n]+tz+breite[n]+tz+laenge[n];
        END;
        WriteLn(^J^M);
(*E*)   FOR n:=3 DOWNTO 1 DO
        WriteLn(n,'.Navigationspunkt: ',navpunkt[n]);
      END.
```

Erläuterungen:

(A) Die Variablen »breite« und »laenge« sind **Arrays** mit bereits vorher festgelegtem Typ »grad«, während das Feld »name« im Variablenteil mit dem Typ **String**[20] versehen wurde.

Im **Array** »navpunkt« fassen wir alle drei Angaben eines Ortes zusammen. Siehe (D).

(B) »tz« ist ein Trennzeichen, das wir in (D) brauchen.

Zum Einlesen der Koordinaten verwenden wir wieder unsere bewährte Prozedur »gradin«.

(C) Eine bestimmte Anzahl gleicher Anweisungen nennt man wiederholte Anweisungen, die man mit der **FOR**-Anweisung bearbeiten kann. Dazu benötigen wir die Zählvariable »n«, die wir in unserem Beispiel nur bis zum Wert 3 hinauftreiben.

Die eigentliche zu wiederholende Anweisung kann ein ganzer Block sein, der zwischen **BEGIN** und **END** steht, oder auch eine einzelne Anweisung, die keine Begrenzer benötigt. Letzteres ist bei (E) verwirklicht.

(D) Hier fassen wir alle Angaben eines Punktes unter einem Namen zusammen, der die gleichen Indizes trägt wie die entsprechenden Einzeldaten.

(E) Die **FOR**-Schleife läßt sich auch in absteigender Weise ausführen. Dazu muß das Wort **TO** durch **DOWNTO** ersetzt werden.

Wenn die Ausgangszahl dabei kleiner ist als die Endzahl, erfolgt kein Durchgang durch die Schleife.

Unabhängig davon, ob mit **TO** oder **DOWNTO** der Zähler verändert wird, erfolgt die Schrittweite immer in ganzen Einheiten. Wer von den BASIC-Dialekten her den Zusatz STEP gewöhnt ist, muß dazu eine eigene Prozedur entwickeln. Turbo hat sie leider nicht implementiert (eingebaut).

Zur Form:

- Die Dimensionierung von **Arrays** muß in eckigen Klammern erfolgen. Der darin angegebene Bereich muß eine endliche Anzahl Elemente aufweisen. Es ist daher z.B. nicht möglich, ein Feld mit **Array**[Integer] anzulegen.

- Allgemein lautet die Schreibweise:

```
Feldbezeichner: Array [von..bis] Of Komponententyp
```

- Die **FOR**-Anweisung benötigt einen Zähler, der aber nicht unbedingt numerisch sein muß. Genauso ist ein Auf- oder Abzählen von Buchstaben, aber auch von

geordneten Farben, Namen, Elementen usw. machbar. Das ist wieder mal Pascal-Freiheit.

- Die eigentliche wiederholte Anweisung beginnt nach einem **DO**, hinter dem kein Strichpunkt stehen darf.

  Nach jedem Durchlauf wird geprüft, ob die Ende-Bedingung, also der angegebene Grenzwert des Zählers, schon erreicht ist. Wenn nicht, wird er erhöht und die Anweisung wiederholt. Wenn ja, erfolgt der Aussprung an das Ende des Blocks.

9.2 Zweidimensionale Arrays (Matrizen)

Schachtelt man **Arrays**, dann erhält man sogenannte Matrizen oder mehrdimensionale Felder. Das wird immer dann notwendig, wenn den Komponenten eines **Arrays** jeweils mehrere weitere Elemente zugeordnet werden müssen. Typisches Beispiel dafür ist das Koordinatensystem, wo man in der Ebene jeder X-Stelle mehrere Y-Werte zuordnen kann. An unserem Beispiel mit den geographischen Koordinaten läßt sich das gut zeigen:

Jedem Ortsnamen werden genau zwei Koordinaten zugeordnet, so daß eine Einheit aus drei Komponenten besteht. Das haben wir oben schon besprochen. Neu ist aber nun die Anordnung:

Wir legen ein Feld an, sagen wir mit vier Namen, das es zuläßt, in jeder Komponente auch noch die geographische Breite und Länge anzuordnen.

Sortiert sieht das nun folgendermaßen aus:

```
Name 1    Breite 1    Länge 1
Name 2    Breite 2    Länge 2
Name 3    Breite 3    Länge 3
Name 4    Breite 4    Länge 4
```

Oder kürzer:

```
Angabe1(name,breite,laenge)
Angabe2(name,breite,laenge)
Angabe3(name,breite,laenge)
Angabe4(name,breite,laenge)
```

Oder noch kürzer:

```
Angabe1(1,2,3)
Angabe2(1,2,3)
Angabe3(1,2,3)
Angabe4(1,2,3)
```

Oder in Pascal-Schreibweise:

```
angabe: Array[1..4] Of Array [1..3] Of bezeichnung;
```

was nichts anderes bedeutet, als daß es vier Feldelemente geben soll, von denen jedes ein Feld mit drei Komponenten ist.

Wiederum wichtig: Alle Komponenten eines Feldes müssen vom gleichen Typ sein, sonst streikt der Turbo-Compiler.

Wie man sich da noch auskennen soll?

Ganz einfach, Sie zählen eben die einzelnen Komponenten ab. Nehmen wir z.B. die Angabe (2,3), dann heißt dies nichts anderes als zweites Feld, dritte Komponente. In unserem Fall ist damit also die geographische Länge des zweiten Orts gemeint.

Insgesamt enthält hier unser Feld »angabe« genau 12 Elemente, jedes davon ist ein String mit einer vorgegebenen einheitlichen Länge. Über die beiden Indizes angesprochen, kann man auf jede Komponente gezielt zugreifen:

```
[1,1],[1,2],[1,3]
[2,1],[2,2],[2,3]
[3,1],[3,2],[3,4]
[4,1],[4,2],[4,3]
```

Jetzt erkennen Sie sicher die Ordnung von oben wieder. Jede Komponente trägt dabei den Namen des Feldes »angabe«, um sie von eventuellen anderen Feldern unterscheiden zu können. Also z.B. »angabe[3,2]«. Das wäre auch bereits die korrekte Pascal-Schreibweise.

Bevor wir uns das nächste Beispielprogramm P18 genauer ansehen, müssen wir noch einen weiteren Begriff klären, den uns Pascal zur Benützung anbietet. Er wird nämlich in P18 verwendet.

9.3 Array-Konstanten

Nachdem Konstanten feste Werte sind, muß eine **Array**-Konstante wohl ein Feld mit bestimmten Werten sein. Und das ist es auch, wobei Turbo großzügig ist und bei typisierten Konstanten neue Belegungen während des Programmlaufs wie bei Variablen zuläßt.

Eine eindimensionale Konstante läßt sich leicht durchschauen. In unserem Beispiel treten doch in geordneter Folge immer der Name eines Ortes, dahinter seine Breite und am Schluß seine Länge auf. Wenn wir diese Begriffe als Kommentare dazuschreiben wollen, bietet sich ein kleines **Array** an, das (geordnet !) die Strings 'Ort: ', 'Breite: ' und 'Länge: ' enthält.

Sie sehen, wir haben schon dafür gesorgt, daß alle Strings von gleicher Länge sind.

Diese Kommentare bilden damit den **Array**-Typ:

```
komment: Array[1..3] Of String[8];
```

Da wir den Kommentar nie ändern wollen, legen ihn wir ein für alle Mal fest mit

```
tex: komment= ('Ort: ','Breite: ', 'Länge: ');
```

Das Ganze kann auch wieder in einer einzigen Zeile geschehen:

```
tex: Array[1..3] Of String[8]=('Ort: ','Breite: ','Länge: ');
```

Und damit haben wir sie doch schon fast im Griff, die **Array**-Konstante. Wir müssen nur noch auf die einzelnen Komponenten zugreifen können. Nichts einfacher als das: tex[2] hat z.B. immer den konstanten Inhalt 'Breite: '. Das ist ja gerade das Nette an den **Arrays**: Alles hat seine Ordnung. So finden wir jedes Element sofort wieder.

Selbstverständlich lassen sich auch **Array**-Konstanten mit mehrdimensionalen Feldern aufbauen. Eine dementsprechende, aber wohl in diesem Zusammenhang kaum brauchbare zweidimensionale Feldkonstante zum Beispiel aus Abschnitt 9.2 hätte demnach 12 Komponenten, die aus vier mal drei Strings bestünden.

Denkbar wäre ein Einsatz dieser besonderen Konstanten, wenn man z.B. in irgendwelchen Streckenberechnungen ständig die Angaben von vier festen Orten benötigt. So etwas könnte dann vielleicht folgende Inhalte haben:

```
fixwert: Array[1..4,1..3] Of String[8]=
(('Astadt  ','46.10.12','10.10.10'),
('Bstadt  ','47.20.24','11.11.11'),
('Cstadt  ','45.30.20','09.08.07'),
('Dstadt  ','47.30.50','12.20.30'));
```

Prüfen Sie selbst nach. Was wäre z.B. »fixwert[2,3]"?

Richtig lustig wird es erst, wenn man **Arrays** mit mehr als zwei Dimensionen zu bearbeiten hat. Sie können sich ja mal den Abschnitt 10.4 anschauen, wenn es Sie interessiert. Doch nun endlich zu unserem Programm P18, das noch eine zusätzliche Neuerung beinhaltet.

9.4 Programm P18: 2-dim-Array, Arraykonstante, geschachtelte FOR-Anweisung

Wir empfehlen Ihnen, dieses Programm zuerst einmal einzugeben und zu starten, damit Sie sehen, was hier eigentlich geschieht. (So eine Untersuchung, die vom Ergebnis ausgeht, nennt man übrigens deduktiv.)

```
      PROGRAM P18; (* Array zweidimensional *)
      TYPE
(*A*)   grad=String[8];
        ort=Array [1..100] Of grad;
(*B*)   bezeichnung=(o,b,l);
        komment=Array[1..3] Of grad;
      VAR
(*C*)   name,breite,laenge:grad;
        angaben:Array [1..100] Of Array [1..3] Of grad;
                (* Array [ort,1..3] Of grad; *)
(*D*)   z,n,i:Integer;
      CONST
(*E*)   tex:komment=('Ort:   ','Breite: ','Laenge: ');
      BEGIN
        ClrScr;WriteLn('Erfassung von Navigationspunkten'^J^J^M);
(*F*)   Write('Wieviele Orte sollen eingelesen werden? ');
        ReadLn(z);
(*G*)   FOR n:=1 TO z DO
        BEGIN
          WriteLn;
(*H*)     FOR i:=1 TO 3 DO
          BEGIN
```

```
(*I*)        Write(tex[i]);
             Read(angaben[n,i]);Write('    ');
           END;
         END;
       END.
```

Die Erläuterungen:

(A) Wir legen die Datentypen für die **Arrays** fest. »komment« ist der Typ für die Feldkonstante »tex«.

(B) »bezeichnung« ist diesmal ein selbstgewählter Typ mit den Komponenten o, b und l für Ort, Breite und Länge. Doch lassen wir das zunächst einmal beiseite.

(C) Erst hier werden die **Arrays** definiert. In Kommentarklammern die Schreibweise, die das vorher definierte **Array** »ort« verwendet.

(D) Das sind die Zählvariablen »n« und »i« für die **FOR**-Schleifen sowie »z« für den oberen Begrenzer.

(E) Jetzt probieren wir natürlich die vorhin gründlich besprochene Feldkonstante »tex« aus.

(F) Diesmal lassen wir uns abfragen, wie viele Orte wir insgesamt einlesen wollen.

(G) Das ist die Einleseschleife für jeweils eine Feldvariable. Wählen wir z.B. für z=4, dann wird sie viermal durchlaufen. Aber jetzt schachteln wir in diese Hauptschleife eine weitere ein:

(H) Diese Schleife liest für jede der vier Feldvariablen jeweils die drei erforderlichen Komponenten ein.

(I) Und damit jedermann weiß, was er eintippen soll, wird mit »tex[i]« der Kommentar mit ausgegeben.

Steht die Zählvariable der Feldvariablen »n« z.B. auf 3 und die Zählvariable »i« der Komponenten z.B. auf 2, dann wird zunächst mit »tex[2]« der String 'Breite: ' geschrieben und danach die Eingabe für die zweite Komponente der dritten Feldvariablen verlangt.

Auf eine weitere Bearbeitung haben wir in diesem Beispiel verzichtet, damit Sie nicht ganz außer Atem kommen. Jedenfalls stehen im Arbeitsspeicher wieder für z Orte die Angaben bereit.

Zur Form:

- Das Wort **Text** sollten Sie möglichst nicht verwenden. Es gehört zu den für Pascal reservierten Wörtern. Deshalb haben wir für die Textkonstante nur »tex« gewählt.

- Komponenten von Feldkonstanten werden zwischen runde Klammern geschrieben und durch Kommata getrennt. Treten in einer Feldkonstanten mehrere Feldkonstanten auf (mehrdimensionale Feldkonstanten), dann werden deren Komponenten ebenfalls wieder in Klammern zusammengefaßt und ebenfalls durch Kommata getrennt (siehe Beispiel »fixwert« in Abschnitt 10.4.)

- Geschachtelte **FOR**-Anweisungen schreibt man zweckmäßigerweise auch im Text geschachtelt. Immer nach der Regel: **END** unter entsprechendem **BEGIN**. Dazwischen eingerückt.

9.5 Frei definierte Datenypen Programm P19: Demo

Das letzte Thema war sicher nicht ganz einfach zu verstehen. Wir sind auch noch nicht ganz fertig damit. Doch an dieser Stelle bietet es sich an, mal etwas pascalianisch zu denken.

Bis jetzt haben wir Sie nämlich weitgehend damit verschont, bei der Deklarierung von Datentypen andere als die Standardtypen **String**, **Integer**, **Real** zu benützen. Das muß sich aber jetzt schnell ändern, sonst kommen Sie nicht weg vom starren BASIC-Denken.

Falls die folgenden Ausführungen jetzt für Sie noch etwas zu früh kommen, überspringen Sie sie einfach. Das nächste Kapitel fängt dafür wieder recht gemütlich an. Wir wollen Sie nicht unbedingt zum Wahnsinn treiben.

In Programm 18 haben wir bei (B) schon ganz leise angedeutet, daß man einen Datentyp z.B. auch mit (o,b,l) definieren kann. Nennen wir also diesen Typ, in dem die Elemente einzeln aufgezählt werden, einfach mal »bezeichnung«.

Dann läßt sich damit auch ein Feldtyp erstellen, der z.B. »kom« genannt wird und vom Typ »**Array** [bezeichnung] **Of** grad« ist. Dieses Feld besteht demnach aus drei Feldvariablen. Wie sie genannt werden, liegt noch nicht fest. Das geschieht erst bei der Variablen- oder der Konstantendeklarierung.

Zusammengefaßt ergibt sich ein Datentyp der Form

```
kom= Array [bezeichnung] Of grad;
```

Wer sagt eigentlich, daß man nur mit Zahlen eine **FOR**-Anweisung ordentlich (das ist wörtlich gemeint) durchlaufen kann? Genausogut kann man eine Variable »j« hernehmen, die eben diesmal nicht vom Typ **Integer** ist, sondern von unserem selbstgewählten Typ »bezeichnung«. Das tun wir im **VAR**-Teil mit

```
j: bezeichnung;
```

Um das Ganze noch ein bißchen zu verschönern, deklarieren wir auch noch eine Feldkonstante – wozu haben wir das vorhin besprochen - mit dem eben erklärten Typ »kom«.

Sie ahnen schon, was jetzt im Teil **CONST** kommt:

```
tex: kom= ('Ort:     ','Breite: ','Länge: ');
```

Der Witz des Ganzen kommt aber erst im Anweisungsteil bei der inneren **FOR**-Schleife: Ohne Rücksicht auf bisherige Programmierkonventionen, die uns u.a. BASIC auferlegt hat, schreiben wir

```
FOR j:= o TO l DO
```

Natürlich muß der Index von »tex« jetzt ebenfalls »j« sein. Und auch bei den »angaben« gibt es keinen Zähler »i« mehr, auch der heißt jetzt »j«. Wir zählen einfach mit den Elementen, die der Typ »bezeichnung« hergibt. Und das sind eben – wohlgeordnet – die Buchstaben o, b und l. Sie bilden die Indizes für Ort, Breite und Länge.

Wenn wir jetzt unser abgeändertes Programm P18 noch einmal anschauen, dann müßte uns bei der Deklaration des zweidimensionalen Feldes »angaben« etwas eigenartig vorkommen: Es darf jetzt eigentlich kein "**Array** [1..3] **Of** grad« mehr geben, denn wir zählen ja die Komponenten gar nicht mehr von 1 bis 3.

Werfen wir also schnell die alten Zähler hinaus, und ersetzen sie durch unsere neuen.

Dabei heißt es aufpassen, denn Turbo akzeptiert die Form **Array** [o,b,l] nicht und schon gar nicht [o..l]. Denn woher sollte der Compiler wissen, was wir zwischen o und l erwarten? Deswegen müssen wir auf den vordefinierten Typ »bezeichnung" zurückgreifen und schreiben:

```
angaben: Array [1..100] Of Array [bezeichnung] Of grad;
```

Jetzt ist auch Turbo zufrieden.

Es ist sicher verständlich, daß Sie jetzt äußerst mißtrauisch geworden sind, ob so etwas auch tatsächlich funktioniert. Sie haben durchaus recht: Diesmal wollen wir unser Programm auch durch eine Bildschirmausgabe kontrollieren.

Hängen wir deswegen noch eine geschachtelte **FOR**-Anweisung dran, und lassen wir uns die Eingaben formatiert wieder ausspucken.

Vorsichtshalber drucken wir das veränderte Programm P18 noch einmal komplett ab, damit Sie uns keinen Vorwurf machen können, wenn nicht alles so läuft, wie wir das eben durchgeackert haben. Nennen wir die abgewandelte Form Programm P19.

Wir haben dabei auf jeglichen Anwenderkomfort wie Setzen von **BufLen**, Verwendung der Include-Prozedur »gradin« usw. verzichtet. Diesmal kommt es uns nur auf das Verständnis für die selbstdefinierten Datentypen an.

Machen Sie deshalb durch vielfaches Abwandeln von P19 von den Turbo-Möglichkeiten ausgiebig Gebrauch. Und nun viel Vergnügen, das Sie nur haben werden, wenn Sie sich bei unseren vorherigen Gedankengängen ordentlich haben stressen lassen. Denn diesmal gibt's keine zusätzlichen Erläuterungen mehr:

```
PROGRAM P19; (* Array zweidimensional *)
TYPE
  grad=String[8];
  bezeichnung=(o,b,l);
  kom=Array [bezeichnung] Of grad;
VAR
  name,breite,laenge:grad;
  angaben:Array [1..100] Of Array [bezeichnung] Of grad;
  {oder kurz: angaben:Array [1..100] Of kom}
  z,n:Integer;
  j:bezeichnung;
CONST
  tex:kom=('Ort:    ','Breite: ','Laenge: ');
BEGIN
```

```
  ClrScr;
  Write('Wieviele Orte sollen eingelesen werden? ');
  ReadLn(z);
  FOR n:=1 TO z DO
  BEGIN
    WriteLn;
    FOR j:=o TO l DO
    BEGIN
      Write(tex[j]);
      Read(angaben[n,j]);Write ('    ');
    END;
  END;
  WriteLn(^J^J^J^M'Folgende Daten im Array erfasst:',^J^M);
  FOR n:=1 TO z DO
  BEGIN
    Write(^J^M,n,'.');
    FOR j:=o TO l DO
    Write(tex[j],angaben[n,j]:10, '     ');
  END;
END.
```

Kapitel

10

Zauberwürfel, Datenstrukturen und mehrdimensionale Felder

10.1 Der Zauberwürfel in verschiedenen Dimensionen

Wir haben im vorigen Kapitel schon erkannt, daß sich Daten wesentlich leichter weiterverarbeiten lassen, wenn sie nicht nur gesammelt in einem gemeinsamen Topf untergebracht werden, sondern wenn innerhalb der Daten bereits eine von uns selbst festgelegte Ordnung herrscht.

Diese Arbeit mit Datenstrukturen werden wir nun an einem recht interessanten und einleuchtenden Beispiel genauer unter die Lupe nehmen. Dazu verlassen wir vorübergehend unser Thema »Koordinaten« und spielen mal ein bißchen mit Körpern und deren Vertauschungen.

Sie kennen sicher den Zauberwürfel, der sich durch Verdrehen aus allen möglichen Lagen wieder in seine Ausgangslage zurückführen läßt, so daß alle Flächen eine einheit-

liche Farbe aufweisen. Normalerweise besteht er aus 27 Miniwürfeln, die untereinander durch besondere Gelenke verbunden sind.

Wir wollen nun versuchen, so einen Würfel zu simulieren, allerdings beschränken wir uns auf eine Seitenlänge von nur 2 Einheiten, so daß insgesamt 8 Miniwürfel den Gesamtkörper bilden. Wer Spaß daran hat – und den gibt es bei diesem Spielchen bestimmt – , der kann das Gelernte ja hinterher mit dem größeren Bruder unseres Versuchsobjekts nachvollziehen; denn an der grundsätzlichen Arbeitsweise ändert sich nichts.

Um die Lage der Miniwürfel in jeder Konfiguration feststellen zu können, müssen wir sie ordnen und benennen. Dazu bieten sich Variablennamen an. Die einfachste Lösung für die Benennung ist eine Numerierung von 1 bis 8 mit Hilfe von einfachen Variablen, also z.B. »nr1«, »nr2«, ..., »nr8«. Doch wie halten wir die Position der Einzelteile fest?

Denkbar wäre, daß wir unsere Variablen als Strings definieren und den Ausgangszustand folgendermaßen beschreiben:

```
nr1:='oben li vorn'
nr2:='oben re vorn'  usw.
```

Der Würfel könnte dann so angeordnet sein:

```
     3------4
1------2    |
|    .  |   |
|    .  |   |
|    7..|...8
5------6
```

Doch sobald wir zu verdrehen anfangen, beispielsweise die rechte Schicht nach hinten (im Uhrzeigersinn), gibt es jede Menge zu tun, um die neue Zusammenstellung festzuhalten:

Variable »nr2« muß nun mit 'oben rechts vorn' belegt werden usw. Und wie wollen wir überprüfen, ob der Körper seine Ausgangslage hat oder nicht?

Selbstverständlich geht das mit viel Aufwand, aber das ist nicht unsere Absicht. Lassen wir also mal diese skalaren Daten beiseite und erinnern uns, daß es auch Felder gibt.

10.2 Miniwürfel als eindimensionales Feld Programm P20: selbst definierte Datentypen, Ord

Belassen wir die Anordnung so, wie wir sie oben skizziert haben, dann kann ein **Array** aus 8 Komponenten schon etwas Ordnung schaffen, denn es hat in sich ja schon eine abzählbare Ordnung. Wählen wir also ein **Array** [m1..m8] **Of** mini, dann genügt eine einzige Variable »wuerfel«, um auf jeden Miniwürfel zugreifen zu können.

Erstellen wir also mal ein kleines Programm, das alle Würfelchen erfaßt:

```
      PROGRAM P20; (* Strukturen *)
      TYPE
(*A*)   mini=(m1,m2,m3,m4,m5,m6,m7,m8);
(*B*) VAR
        wuerfel:Array [m1..m8] Of mini;
        i:mini;
(*C*) BEGIN
        ClrScr;
        WriteLn('Miniwuerfel eindimensional gegliedert:'^J^J^M);
        FOR i:=m1 TO m8 DO
        BEGIN
          wuerfel[i]:=i;
(*D*)     WriteLn(Ord(i)+1);
        END;
      END.
```

Erläuterungen:

(A) Als Datentyp legen wir »mini« fest, mit aufgezählten Datenelementen »m1« bis »m8«. Dies macht Pascal möglich. Aber Achtung: Mit diesen Typen kann man nicht so ohne weiteres Rechenoperationen durchführen oder sie gar auf dem Bildschirm ausdrucken. Es handelt sich hier nämlich weder um Strings noch um numerische Daten, sondern um unsere eigenen »mini«-Werte.

(B) Das **Array** vom Typ »mini« läßt sich nun aber trotzdem abzählbar gestalten, weil bei der Typdeklarierung durch die Aufzählung der Daten eine Reihenfolge festgelegt wurde. Diese übernehmen wir in das **Array**.

Achtung: Es ist hier nicht möglich, **Array** [1..8] zu verwenden, weil wir eben keine numerische Abzählung vornehmen.

Auch einen Zähler »i« benötigen wir vom Typ »mini«. Wem das etwas sonderbar vorkommt, der hat wahrscheinlich viel zu lange mit BASIC programmiert.

(C) Der Hauptteil spricht eigentlich für sich: Wir weisen mit Hilfe einer **FOR**-Schleife jeder Komponente des **Arrays** eine Miniwürfelbezeichnung zu, denn bis jetzt war das Feld »wuerfel« ja noch ohne Inhalt. Die Typbezeichnung **Of** mini hatte noch gar nichts bewirkt.

(D) Um zu überprüfen, was wir da gerade zugeordnet haben, können wir nicht auf ein **WriteLn**(wuerfel[i]) oder ähnliches zurückgreifen, denn solche Ausgaben sind in Turbo nicht vorgesehen, was der Comiler auch reklamiert, wenn es trotzdem versucht wird.

Bleibt eine andere Möglichkeit: Wir lassen uns die Ordnungszahl des jeweiligen Elements ausgeben. Bei der Zuweisung von »m3« ist das die Nummer 2, weil Turbo sich an die Computervereinbarung hält und mit 0 zu zählen beginnt. Deswegen müssen wir immer noch eins dazuzählen, wenn wir die Nummer des Miniwürfelchens sehen wollen, denn unsere vereinbarte Zählweise geht von 1 bis 8. Der entsprechende Befehl zur Feststellung dieser Zahl ist der Aufruf der Standardfunktion **Ord** mit der zu überprüfenden Variablen in runden Klammern.

Achtung: **Write**(**Ord**(m3)) erzeugt ebenfalls den Ausdruck »2«, weil m1 bis m8 wie Variablenwerte gehandelt werden.

So weit, so gut. Doch z.B. bleibt die Position von »m6« innerhalb des Gesamtkörpers zunächst noch etwas undurchsichtig. Wir müssen dazu erst einmal die Zeichnung anschauen. Auch der Programmstart bringt nur die Folge »1« bis »8« auf den Schirm. Mit einem Würfel hat das noch nicht viel Ähnlichkeit.

Die Zuordung beim Verdrehen des Gebildes wird aber schon etwas einfacher: m6 kommt dorthin, wo m2 war, m8, wo m6 war usw.

Da ließe sich schon ein Algorithmus finden, der alle Möglichkeiten erfaßt. Doch beim Drehen einer weiteren Ebene stoßen wir auf erhebliche Schwierigkeiten, die Übersicht zu behalten. Das wollen wir an dieser Stelle auch gar nicht ausprobieren.

10.3 Versuch mit einem zweidimensionalen Array Programm P21: geschachteltes FOR, Retyping

Jetzt gehen wir weg von der einfachen Aufzählordnung, die uns ein eindimensionales Feld liefert, und strukturieren eine Stufe weiter. Wenn wir uns den Würfel genauer betrachten, zergliedert er sich nämlich in lauter 2er-Reihen. Genauer: Vier solche Reihen bilden den Gesamtkörper.

Was liegt näher, als ein **Array** zu bilden, das aus lauter »Unterfeldern« zu je zwei Miniwürfeln besteht:

```
Array [1..4] Of Array [1..2] Of mini
```

Zum Abzählen haben wir diesmal wieder die üblichen Integerzahlen hergenommen. Das schont unsere Nerven etwas.

Die Zuweisung der Miniwürfel-Bezeichnungen kann jetzt in einer geschachtelten Schleife erfolgen, wobei die äußere Schleife die höherwertige Dimension »reihe« zählt, die innere, niederwertigere zählt die Würfelchen immer schön paarweise ab.

Schauen wir uns das Pascal-Programm dazu an:

```
      PROGRAM P21; (* Zweidimensional *)
(*A*) TYPE
        mini=(m1,m2,m3,m4,m5,m6,m7,m8);
        reihe=Array [1..2] Of mini;
      VAR
(*B*)   wuerfel2:Array [1..4] Of reihe;
        ri,mi:Integer;
      BEGIN
        ClrScr;
        WriteLn('Miniwuerfel zweidimensional gegliedert:'^J^J^M);
(*C*)   FOR ri:=0 TO 3 DO
        FOR mi:=0 TO 1 DO
(*D*)   wuerfel2[ri+1,mi+1]:=mini(ri*2+mi);
(*E*)   FOR ri:=1 TO 4 DO
        BEGIN
          WriteLn;
          FOR mi:=1 TO 2 DO
          Write(' ',Ord(wuerfel2[ri,mi])+1);
        END;
      END.
```

Erläuterungen:

(A) Die Minis bleiben zunächst, wie sie sind. Aber wir definieren gleich die »reihe« vor und sparen uns das bei der Variablen-Deklarierung.

(B) »wuerfel2« soll andeuten, daß wir nun mit zwei Dimensionen arbeiten: Die Variable stellt ein Feld aus Feldern dar.

Für die geschachtelte Schleife brauchen wir zwei Zähler, einen für die Reihen und einen für die Minis.

(C) Warum sollen wir unsere 4 Reihen nicht auch mal von 0 bis 3 zählen? Allerdings muß dann bei der Numerierung jeweils 1 addiert werden, weil wir uns im **VAR**-Teil auf die Folge 1..4 festgelegt haben.

Bei der Zuweisung der Werte dagegen ist der Beginn mit 0 ganz praktisch, da der Typ »mini« von 0 aufwärts zählt. Eine Korrektur ist nicht notwendig.

Anmerkung: Zweckmäßigerweise bleibt man beim Programmieren eines solchen Problems bei einer, einmal vorgewählten Zählweise und ändert sie nur im Notfall. In unserem Fall wäre es zum Beispiel richtig, bei der Typdefinition von »mini« als erstes Element ein »Dummy« einzubauen, das nie benützt wird, jedoch erlaubt, das Element m1 als Nummer 1 dieser Datenfolge anzusprechen. Damit kann man bei der konventionellen Zählweise bleiben, ohne daß es größere Probleme gibt.

(D) Das zuzuweisende Element ergibt sich aus der Reihe mal 2 plus Nummer in dieser Reihe. »m5« kommt also auf die zweite Reihe als Nummer 1. Jetzt lohnt es sich wirklich, daß wir mit 0 zu zählen begonnen haben, denn sonst müßten wir zwei weitere »Dummies« einbauen, die verhindern, daß z.B bei Reihe eins Nummer 1 das dritte Element zugeordet wird.

Zählen wir auch noch die paarweisen Minis selbst von 1 bis 2, dann benötigen wir noch eine »Strohfigur«. Oder wir korrigieren jeweils die Zähler um minus 1.

Das mag zur Zählweise genügen, Sie kommen beim Programmieren und bei den Testläufen von alleine dahinter. Denken Sie aber mal an die eben gemachten Bemerkungen, wenn seltsame Ergebnisse erscheinen.

Mit der Zuweisung von mini(ri*2+mi) erfolgt ein sogenanntes Retyping, bei dem aus der Ordnungszahl des Elements wieder das Element selbst aus der

Datenfolge herausgefischt werden kann. Turbo macht das möglich, Standard-Pascal kann dies nicht.

Die Funktion **Ord** und das Retyping, das durch Aufruf des gewünschten Datentyps und Angabe der Ordnungszahl erfolgt, sind demnach jeweils Umkehrfunktionen zueinander.

(E) Die geschachtelte Ausgabeschleife arbeitet mit der eben beschriebenen konventionellen Zählweise, die zum Aufzählen der in Reihen gegliederten Würfelchen paßt. Die Ordnungszahl muß in unserem Fall dabei immer um 1 erhöht werden, weil wir kein Dummy gesetzt haben.

Schon besser, unsere Reihen denken wir uns liegend angeordnet und mit oben links vorn beginnend. Die letzte Reihe mit den Minis »m7« und »m8« liegt dann unten hinten.

Doch beim Drehen der rechten Ebene geht unsere Ordnung schon wieder kaputt, so daß wir bei dieser Anordnung ziemlich kräftige Klimmzüge machen müssen, um die gewünschten Zuordnungen zu programmieren.

Wir sind aber auch selber schuld an unserem Dilemma. Wenn wir schon einen dreidimensionalen Körper analysieren wollen, sollten wir uns nicht mit weniger als drei Dimensionen in unseren **Arrays** begnügen.

10.4 Dreidimensionales Array für dreidimensionalen Körper Programme P22 und P23: Array-Konstante, GotoXY, KeyPressed

Betrachten wir unseren Würfel nun endlich so, wie es ihm zusteht, dann sehen wir, daß immer zwei Miniwürfel eine Reihe bilden und immer zwei Reihen eine Schicht. Bleiben wir bei unserer bisherigen Numerierung, dann ist Schicht Nummer 1 oben, Reihe Nummer 1 vorn und Miniwürfel Nummer 1 links. Und schon haben wir die dreidimensionale Gliederung: Schicht, Reihe, Element.

Halten wir sogleich fest, wie der Körper in geordneter Ausgangslage aussehen soll, dann tun wir das in Turbo mit einer Feldkonstanten, die wir »fixpos« nennen wollen. Sie muß natürlich ebenfalls dreidimensional gegliedert sein, und da man das auf zweidimensionalem Papier nicht ohne weiteres nachvollziehen kann, müssen Klammern herhalten.

Es gilt nun folgende Regel: Die äußersten Klammern umfassen das gesamte **Array**. Die nächstinneren umfassen die Schichten, die wiederum die Reihen aufnehmen. Die Elemente schließlich werden nicht mehr in Klammern geschrieben. Wozu auch, sie stehen immer für sich allein. Alle Komponenten müssen durch Kommata voneinander getrennt werden.

Schauen Sie sich die entsprechende Programmzeile an: In »fixpos« werden immer zwei Elemente mit einer Klammer zusammengefaßt. Das sind die Reihen. Immer zwei Reihen werden mit Klammern zur Darstellung einer Schicht zusammengefaßt.

Diese **Array**-Konstante werden wir dazu verwenden, um festzustellen, ob sich der Zauberwürfel in seiner ursprünglichen Lage befindet.

Bevor wir weiter erklären, zunächst das Programm P22:

```
      PROGRAM P22; (* dreidimesional *)
      TYPE
(*A*)   st=String[1];
        mini=st;
(*B*)   reihe=Array [1..2] Of mini;
        schicht=Array [1..2] Of reihe;
        wuerfel3=Array [1..2] Of schicht;
      VAR
(*C*)   wuerfel:wuerfel3;
        si,ri,mi,ze,sp:Integer;
        nummer,retten:st;
(*D*) CONST
        fixpos:wuerfel3=
        ((('1','2'),('3','4')),(('5','6'),('7','8')));
(*E*) {$I AUSDRUCK.P23}
      BEGIN {Zuweisungen}
(*F*)   FOR si:=1 TO 2 DO
        FOR ri:=1 TO 2 DO
        FOR mi:=1 TO 2 DO
        BEGIN
(*G*)     Str((si-1)*4+(ri-1)*2+(mi-1)+1:1,nummer);
          wuerfel[si,ri,mi]:=nummer;
        END;
        ClrScr;
        ausdruck;
      END.
```

Erläuterungen:

(A) Wir haben diesmal auf die Verwendung eines selbstdefinierten Typs für die Miniwürfel verzichtet und arbeiten dafür mit Strings der Länge 1, die später eine Ziffer aufnehmen.

(B) Das nächsthöhere **Array** wird mit einem bereits deklarierten **Array**-Typ definiert. Wir besorgen die Deklarierung des Typs »wuerfel3« gleich hier.

Achtung: »wuerfel3« ist keine Variable, sondern ein Feldtyp.

(C) »wuerfel« ist die dreidimensionale Feldvariable. Des weiteren benötigen wir nun drei Zähler: für die Schichten, die Reihen und die Elemente.

Zwei weitere Variablen verwenden wir für die Ansteuerung einer bestimmten Bildschirmposition (Spalte, Zeile).

»nummer« und »retten« sind Hilfsvariablen für die Bildschirmausgabe bzw. für die Verdrehung des Würfels.

(D) Diese **Array**-Konstante haben wir oben besprochen. Sie muß Ihnen klar sein, sonst gibt es nachher heftige Verständnisschwierigkeiten. Also im Zweifelsfall noch einmal oben nachlesen!

(E) Die Bildschirmausgabe übergehen wir zunächst. Sie kann später eingefügt werden.

(F) Logischerweise ist die Zuweisungsschleife drei Ebenen tief.

Noch einmal: Die höchstwertige Dimension (bei uns die Schicht) wird als erste bearbeitet. Schließlich fragen wir in der Reihenfolge:

- In welcher Schicht?
- In welcher Reihe?
- Auf welchem Platz?

(G) Die Prozedur **Str** wandelt die Zahl, die sich aus Schicht mal 2 plus Reihe mal 2 plus Platz ergibt, in die Strings von '1' bis '8' um. Sie sehen, daß wir nun mit dem Wert 1 manipulieren müssen, weil wir die Schleifen nicht mit Null beginnend hochzählen.

Achtung: Die Prozedur **Str** darf nicht in **Write** oder **WriteLn**-Anweisungen verwendet werden!

Der Übersicht halber weisen wir das Ergebnis erst einmal der Hilfsvariablen »nummer« zu und dann dem richtigen Element des Feldes »wuerfel«.

Auch hier haben wir mit »:1« noch eine Absicherung eingebaut, damit wirklich nur ein Zeichen aufgenommen wird. Ob das notwendig ist oder nicht, dürfen Sie selbst untersuchen.

Falls Sie das Programm jetzt schon mal ausprobieren wollen, müssen Sie zuerst die Include-Anweisung für den »ausdruck« mit einer Kommentarklammer stillegen. Aber das Ergebnis wird Sie enttäuschen: Die Zuweisung erfolgt sicherlich, aber still und heimlich (aber auch unheimlich schnell, wie sich das für Turbo gehört), so daß der Bildschirm gar nichts zeigt.

Deshalb haben wir die **PROCEDURE** »ausdruck« vorgesehen, die Sie entweder direkt oder über Diskette einbinden können.

Sie besteht aus zwei Teilen:

Der erste Teil erzeugt auf dem Schirm die Anordnung der Miniwürfel, so wie wir sie vereinbart haben. Das heißt, die Ausgangslage sieht so aus:

```
    Nr3 --- Nr4
Nr1 --- Nr2 / |
|           |   |
|           |   |
|           |   |
|   Nr7     | Nr8
Nr5 --- Nr6
```

Drucken wir in der Zuweisungsschleife die Paare (Reihen), wie sie gerade anfallen, dann müßten wir uns den kleinen Zauberwürfel von hinten nach vorn aufgebaut denken. Das ginge zwar genausogut, wir bestehen aber auf der oben skizzierten Anordnung.

Die Prozedur **GotoXY**(sp,ze) unterstützt uns, so daß wir nach der Festlegung einer Anfangsposition Zeile (5+4), Spalte (20-6) zunächst den Ausdruck

```
Nr1    Nr2
```

erhalten.

Dann fahren wir insgesamt zwei Zeilen hoch und vier Spalten nach rechts, um den Text

```
Nr3   Nr4
```

darüberzusetzen.

Der nächste Durchlauf gilt für die zweite Schicht: Also 4 Zeilen runter und wieder 6 Spalten nach links. Der Rest ist wie beim Ausdruck der oberen Schicht.

Wenn Sie das verwirren sollte, probieren Sie Ihr eigenes System. Wir wollen hier weder großartige Grafik noch ausgefeilte räumliche Darstellung produzieren, sondern uns lediglich ein Bild von unserem Würfel machen. Probieren Sie auch hier ein wenig selbst herum.

```
PROCEDURE (* ausdruck.p23 - Teil 1 *) ausdruck;
BEGIN
  ze:=5;sp:=20;
  FOR si:=1 TO 2 DO
  BEGIN
    ze:=ze+4; sp:=sp-6;
    FOR ri:=1 TO 2 DO
    BEGIN
      sp:=sp+2*ri;ze:=ze-ri+1;
      GotoXY(sp,ze);
      FOR mi:=1 TO 2 DO
      Write('  Nr',wuerfel[si,ri,mi]);
    END;
  END;
```

Der zweite Teil der **PROCEDURE** »ausdruck« dient dazu, die Lage der einzelnen Miniwürfel mit ihrer Ausgangslage zu vergleichen. Auf diese Weise läßt sich später feststellen, ob das Ziel erreicht wurde, durch Verdrehen der einzelnen Ebenen wieder die Urform herzustellen.

Die entscheidende Zeile steht bei (A). Für jede Schicht wird in jeder Reihe jedes Element mit »wuerfel(si,ri,mi)« eingelesen und mit dem entsprechenden Element aus dem fest vorgegebenen **Array** »fixpos« verglichen. Stimmen beide überein, erfolgt der Ausdruck 'hat Ausgangslage' andernfalls (**ELSE**) eben 'nicht ...'.

```
        (* Teil 2 *)
        WriteLn(^J^J);
        FOR si:=1 TO 2 DO
        FOR ri:=1 TO 2 DO
        FOR mi:=1 TO 2 DO
        BEGIN
          Write('Nr.',wuerfel[si,ri,mi],' in Schicht ',
                si,' Reihe ',ri,' Nummer ',mi,': ');
(*A*)     IF wuerfel[si,ri,mi]=fixpos[si,ri,mi]
          THEN WriteLn('Ausgangslage        ')
          ELSE WriteLn('nicht Ausgangslage!');
        END;
      END;
```

Es steht Ihnen frei, auf diese Überprüfungen zu verzichten, aber zum Ausprobieren unseres Programms ist dieser Teil recht brauchbar. Sie können die Überprüfung so programmieren, daß nur dann eine Anzeige erscheint, wenn alle Minis an ihrem Platz sind.

Nachdem Sie diese Prozedur im Programm untergebracht haben, kann der Testlauf erfolgen. Sie erhalten die Würfelanordnung mit Hilfe der Nummernbezeichnung und darunter für jedes Element die Aussage, daß es an seinem ursprünglichen Platz ist.

10.5 Es darf verdreht werden Programm P24/P25: KeyPressed

Was jetzt kommt, ist erst das, was richtig Spaß macht: Wir verdrehen die Ebenen des Würfels. Dazu ein paar Vorüberlegungen:

Wir gehen davon aus, daß der Würfel insgesamt nicht verdreht wird, sondern daß immer nur eine Ebene ihre Lage ändert. Dazu gibt es genau sechs zur Auswahl, weil ein Würfel eben sechs Flächen hat. Immer die Minis, die an einer Fläche hängen, können um den Mittelpunkt dieser Fläche rotieren. Eine Dreheinheit ist dabei 90 Grad (Vierteldrehung).

Eine Ebene kann im Uhrzeigersinn oder entgegen dieser Richtung verdreht werden. Aber diese beiden Möglichkeiten brauchen wir nicht extra vorzusehen, denn Drehung um 90 Grad nach links bedeutet, die gleiche Lage herzustellen wie mit dreimaliger Drehung um 90 Grad nach rechts. (Aha, dreimaliger Aufruf der entsprechenden Prozedur!)

Um die Drehung zu simulieren, müssen die Nummern der einzelnen betroffenen Minis ausgetauscht werden. Das ist ein Ringtausch, eine Art Permutation, von jeweils vier Werten.

Dazu fangen wir an einer beliebigen Stelle an und retten den darin enthaltenen Wert in die dafür vorgesehene Variable »retten»; denn wenn wir sofort den benachbarten Wert eintauschten, wäre der ursprüngliche überschrieben. Den brauchen wir aber, um die letzte freiwerdende Position zu besetzen.

Für zwei Drehungen bieten wir Ihnen den Pascal-Text, die anderen vier schaffen Sie dann sicher selbst. Beachten Sie dabei auch, daß immer ein Index im Feld »wuerfel« einen konstanten Wert beibehält. Ist ja klar: Wenn ich die obere Schicht verdrehe, bleibt die untere unverändert. Verdrehe ich die linke Ebene, betrifft das nur alle ersten Minis usw.

```
(* Drehung der rechten Ebene im Uhrzeigersinn *)
PROCEDURE (* P24 *) dreh_rechts;
BEGIN
  retten:=wuerfel[1,1,2];
  wuerfel[1,1,2]:=wuerfel[2,1,2];
  wuerfel[2,1,2]:=wuerfel[2,2,2];
  wuerfel[2,2,2]:=wuerfel[1,2,2];
  wuerfel[1,2,2]:=retten;
  ausdruck;
END;

(* Drehung der oberen Ebene im Uhrzeigersinn *)
PROCEDURE (* P25 *) dreh_oben;
BEGIN
  retten:=wuerfel[1,1,1];
  wuerfel[1,1,1]:=wuerfel[1,1,2];
  wuerfel[1,1,2]:=wuerfel[1,2,2];
  wuerfel[1,2,2]:=wuerfel[1,2,1];
  wuerfel[1,2,1]:=retten;
  ausdruck;
END;
```

Den noch fehlenden Prozeduren geben Sie die entsprechenden Namen »dreh_unten«, »dreh_links«, »dreh_vorn« und »dreh_hinten«.

Wenn Sie jetzt noch aus dem Zuweisungsteil eine Prozedur machen, indem Sie diesen Teil mit **PROCEDURE** zuweisung; überschreiben und als erste Prozedur hinter den Deklarierungsteil setzen, dann können Sie das Hauptprogramm folgendermaßen ausbauen:

```
        (* P24/P25 *)
        BEGIN
        zuweisung;
        ausdruck;
(*A*)     REPEAT UNTIL KeyPressed;
            dreh_rechts;
          REPEAT UNTIL KeyPressed;
            dreh_oben;
          REPEAT UNTIL KeyPressed;
           { ...  usw ... }
        END.
```

Das ist zwar für die Steuerung unserer Drehung völlig unzureichend, weil eine feste Abfolge programmiert wurde, aber zum Ausprobieren und zur eventuellen Fehlersuche ist es zunächst ausreichend.

Erläuterungen:

(A) Die Standardfunktion **KeyPressed** ist eine sogenannte **Boolean**-Funktion, die zwischen wahr (true) und falsch (false) entscheidet. **True** und **False** sind vordefinierte Bezeichner (Namen) für diese sogenannten Wahrheitswerte. Bei **REPEAT UNTIL KeyPressed** wird die Schleife so lange durchlaufen, bis es wahr ist, daß eine Taste gedrückt wurde. **KeyPressed** hat dann den Wert **True**. Das **UNTIL** bezieht sich immer auf einen Wahrheitsgehalt.

Der Sinn dieser Anweisung in unserem Programm ist nun klar: Wir wollen verhindern, daß alle eingebauten Drehungen ohne Halt über den Bildschirm scrollen; denn wir möchten ja schließlich überprüfen, ob die Drehungen auch richtig ausgeführt worden sind.

Beachten Sie aber bitte, daß hier bereits Drehungen verknüpft werden, und zwar durch Hintereinanderausführung. Im beschriebenen Fall wird zuerst eine Drehung der rechten Seite durchgeführt. Die folgende Anweisung »dreh_oben« bezieht sich nun nicht mehr auf die Ausgangslage, sondern auf die Konfiguration, in der der Würfel nach der ersten Veränderung zur Ruhe kommt.

Sie ahnen schon, was als nächstes kommt?

Jawohl, wir wollen nun frei entscheiden können, welche Ebene in welche Richtung gedreht werden soll.

Kapitel

11

Ein Turbo-Menü

11.1 Was darf man denn wählen dürfen?

Bleiben wir noch bei unserem Würfel, und setzen wir unsere bisher erarbeiteten Bausteine zu einem kompletten Programm zusammen. Dazu gehört in unserem Fall eine Auswahl an Tasten, die es uns erlaubt, die sechs Ebenen per Tastendruck beliebig zu verdrehen.

Dazu sehen wir die Eingaben vor, die man sich mit Hilfe der Anfangsbuchstaben leicht merken kann: **<O>** für oben, **<U>** für unten usw.

Außerdem gebietet es der Anstand, daß man dem Anwender Gelegenheit gibt, das Programm jederzeit verlassen zu können, ohne daß erst der Hauptschalter betätigt werden muß. Also lassen wir auch die Eingabe der Taste **<E>** zu, die für »Ende« steht.

Nun kann es sein, daß man allein mit dem Verstellen des Würfels nicht zufrieden ist. Der Mensch braucht eine Aufgabe: Wir geben ihm diese über die Taste **<Z>** an die Hand. Drückt er sie, dann liefert ihm unser Programm eine Würfelstellung, die er wieder in die Ausgangslage zurückführen muß oder darf. Den entsprechenden Programmteil müssen wir aber erst noch erarbeiten.

Als letzten Auswahlpunkt bieten wir noch die Taste **<A>** an, mit der man sich die ursprünglich als Ausgangskonfiguration definierte Stellung unserer Miniwürfel auf dem Bildschirm anschauen kann.

11.2 Eine Menge Tasten Programm P26: Set Of, IN, IF ... THEN ... ELSE

Fassen wir die für unser Würfelspiel benötigten Tasten zusammen, dann sind dies:

<L>, <R>, <O>, <U>, <V>, <H> für die Drehungen der sechs Ebenen (Flächen) und

<E>, <A>, <Z> für 'Ende', 'Ausgangsstellung' und die Ausgabe einer beliebigen Stellung per 'Zufall'.

Beim Eintippen einer dieser Möglichkeiten lassen sich diese Tasten mit Hilfe ihrer Zeichen (Characters) unterscheiden. Turbo gibt uns dazu noch die Möglichkeit, alle verwendbaren Zeichen in einer Menge zusammenzufassen.

Wie in der Mathematik müssen wir aber erst einmal festlegen, in welcher Grundmenge wir arbeiten wollen. Das geschieht bei der Deklarierung des Datentyps. Nennen wir ihn einmal »zeichen«. Der eigentliche Inhalt unserer Menge kann dann mit Hilfe einer Variablen aufgenommen werden, die natürlich vom Typ »zeichen« sein muß.

Wer jetzt frustriert das Buch zuschlagen will, weil er mit dem Wort Menge böse Erinnerungen an seine schulmathematische Zeit verknüpft, möge bitte noch etwas warten. Nach wenigen Zeilen hat sich unser Problem in Wohlgefallen aufgelöst, wenn wir dazu den Ausschnitt aus einem Programmteil betrachten, den wir später noch verwenden wollen:

```
        PROGRAM P26; (* Mengendemo *)
        (* Mengendefinition, Typ, Variable, Teilmenge *)
        TYPE
(*A*)     zeichen=Set Of Char;
        VAR
(*B*)     auswahl:zeichen;
        BEGIN
          WriteLn('Elemente: O,U,R,L sind in  Menge enthalten',^J^M);
(*C*)     auswahl:=['O','U','R','L'];
(*D*)     IF 'R' IN auswahl THEN WriteLn('R ist dabei.')
          ELSE WriteLn('R ist nicht dabei.');
```

```
  IF 'X' IN auswahl THEN WriteLn('X ist dabei.')
  ELSE WriteLn('X ist nicht dabei.');
END.
```

Erläuterungen:

(A) Ein neuer Datentyp ist nun die Menge, die mit **SetOf** ... beschrieben wird. Hinter dem **Of** muß ein einfacher Datentyp stehen, der den Typ der einzelnen Elemente der Menge angibt.

Damit ist schon eines klar: Alle Elemente einer Menge sind grundsätzlich vom gleichen Typ. In unserem Fall wählen wir dafür die ASCII-Zeichen, die der Computer kennt. Darunter sind nun auch alle Ziffern und sonstige Zeichen wie Komma, Dollar, Doppelkreuz usw.

Wir können natürlich auch – nachdem wir ohnehin nur auf Buchstaben Wert legen – von vornherein bei der Typfestlegung sagen, daß wir nur die Buchstaben zulassen wollen.

Dann können wir den Typ so festlegen:

```
zeichen=Set Of 'A'..'Z';   ( ıur Großbuchstaben) oder
zeichen=Set Of 'A'..'z';   ( 'roß- und Kleinbuchstaben)
```

Dabei haben Sie bei den Datentypen wiederum freie Wahl. Wenn Sie mit Farben oder Lagen arbeiten wollen, dann tun Sie das z.B. so:

```
lagen=Set Of (oben,unten,links,rechts,hinten,vorn,vlinks);
```

Maximal können Sie 256 Elemente in einer Menge erfassen. Mehr läßt auch Turbo nicht zu. Geordnete, also abzählbare Datentypen wie der Grundtyp **Char**, können bereichsweise angegeben werden mit den Anfangs- und Endbegrenzern und zwei Punkten dazwischen, wie wir das mit dem Typ »zeichen« gerade getan haben.

(B) Mit dem Bezeichner »zeichen« läßt sich aber im Programm noch nichts anfangen. Und hier ergeben sich meistens die Verständnisschwierigkeiten bei Pascal: Verarbeitet können immer nur Daten werden, nicht ihre Typen. Daten können aber durch Variablen vertreten werden.

Erlauben Sie uns dazu einen Vergleich aus dem Alltag:

Eine Computerfirma braucht Programmierer. Das sind die Typen, die man aus einer Menge, einem '**Set Of** Leute' definieren kann. Wobei der Typ »Leute« ein Grundtyp sein soll:

```
Programmierer=Set Of Leute;
```

Mit diesem Sachverhalt allein wird die Firma wohl nie ein Programm verkaufen können. Es müssen auch Arbeitsplätze vorhanden sein, auf denen »Leute« vom Typ »Programmierer« sitzen. Erfaßt man alle Arbeitsplätze für Turbo-Spezialisten zusammen, so müssen diese natürlich von »Leuten« des Typs »Programmierer« besetzt sein. In Pascal:

```
turbospezi: programmierer;
```

Noch immer macht diese Firma keinen Umsatz, denn die Arbeitsplätze müssen auch besetzt sein, und erst wenn die Belegschaft beisammen ist, kann man ihr einen sinnvollen Auftrag erteilen. Das könnte das Team sein:

```
turbospezi:=['Meier','Moser','Huber');
```

Genauso gehen wir in unserem Programm vor:

- Datentyp definieren
- Variable festlegen
- Werte zuweisen (Daten bearbeiten)

(C) Bei der Zuweisung der einzelnen Zeichen wird nun mit der Variablen »auswahl« eine Menge festgelegt. Jetzt erst liegen also die verwertbaren Daten vor.

Wir fügen als Test eine kleine Überprüfung an, wobei wir noch einmal auf die Sequenz **IF..THEN..ELSE** hinweisen. Was als Ergebnis auf dem Bildschirm erscheint, dürfte klar sein, oder?

(D) Ob ein Element tatsächlich in einer Menge vorhanden ist, läßt sich mit **In** überprüfen.

11.3 Tastaturabfrage, Programm P27: KBD, UpCase, CASE OF, Ord, ASCII-Code

Bis jetzt können wir überprüfen, ob sich ein gewünschtes Element in einer Menge befindet oder nicht. Wir gehen noch einen Schritt weiter: Wir wollen keinen Unterschied machen zwischen Groß- und Kleinbuchstaben. Das heißt, es soll egal sein, ob der Spieler ein kleines oder ein großes »O« eintippt, wenn er die obere Ebene verdrehen will.

Turbo sieht für solche Fälle die Funktion **UpCase** vor, die aus einem Kleinbuchstaben einen geshifteten (Groß-)Buchstaben macht. Liegt bereits ein Großbuchstabe vor, passiert gar nichts, er bleibt, wie er ist.

Aber wir bauen noch eine Raffinesse ein: Wenn tatsächlich ein großer, mit **<SHIFT>** eingegebener Buchstabe aufgenommen wurde, soll die Drehung nach links, also entgegen dem Uhrzeigersinn, statt nach rechts durchgeführt werden. Dazu muß man nur wissen, daß die Kleinbuchstaben in der ASCII-Reihenfolge immer einen größeren Code als 96 haben, während die Großbuchstaben immer darunter liegen. Siehe dazu die Tabelle im Anhang D.

Wieder ein kleines Programm zum Ausprobieren:

```
      PROGRAM (* P27 *) menu2;
      VAR
(*A*)   auswahl:Set Of Char;
        ch:Char;
        BEGIN
        WriteLn('O,U,R,L geben einen Ton.');
        Write('Taste druecken: ');
        auswahl:=['O','U','R','L'];
        REPEAT
(*B*)     Read(KBD,ch);
(*C*)     IF UpCase(ch) In auswahl THEN Write(^G);
(*D*)   UNTIL ch='E';
        ClrScr;
        Write(^J^J^J'Test-Programm beendet.');
      END.
```

Erläuterungen:

(A) Der Datentyp kann auch im Variablenteil deklariert werden.

(B) Wir lesen über die Datei **KBD** (Tastatur) ein und verhindern damit ein Bildschirmecho. Es würde beim Würfelspiel stören, wenn jedesmal, wenn wir die Taste **<O>** drücken, um die obere Ebene zu verdrehen, auch immer ein »o« auf dem Bildschirm erscheint.

(C) Übersetzt heißt diese Zeile: Wenn der entsprechende Großbuchstabe der eingetippten Taste ein Zeichen ist, das in der Menge »auswahl« enthalten ist, dann soll der eingebaute Klingelton (Kontrollcode ^G) ertönen.

(D) Damit Sie dies einige Male probieren können, haben wir dafür eine **REPEAT**-Schleife gewählt, die Sie nur mit einem großen **<E>** verlassen können, denn das **UpCase** bezieht sich nur auf die Überprüfung in der Zeile darüber!

Übrigens muß das »E« nicht unbedingt in »auswahl« enthalten zu sein. Die Variable »ch« nimmt nämlich alle Daten vom Typ **Char** an, das haben wir schließlich im **VAR**-Teil so ausgemacht.

Bitte zu beachten: Es wäre nicht möglich (bzw. sinnvoll), die Variablen folgendermaßen zu deklarieren:

```
auswahl,ch: Set Of Char;
```

»ch« wäre dann nämlich ebenfalls eine Menge. Und das wollen wir hier doch lieber nicht zulassen.

11.4 Aufbau des Menüs, Programm P28: WHILE..DO, CASE..OF, UpCase,Exit, Label

Jetzt haben wir bis auf ein paar Kleinigkeiten alle Zutaten bereit, um unser Menü zusammenzustellen. Wir binden es ein in das Hauptprogramm, das den Würfel bedienen soll.

Dazu benützen wir eine bisher noch nicht verwendete Art von Schleife. Sie beginnt mit **WHILE**..., was soviel heißt wie 'so lange...'. Dann folgt eine Bedingung, danach das Wort **DO**... für 'mach folgendes:...'

Da in unserem Beispiel so lange auf eine Tastatureingabe gewartet wird, bis der Rechner das erlösende **<E>** für Ende erhält, starten wir mit der Anweisung: »Arbeite den Menü-Block so lange ab, solange die gedrückte Taste nicht **<E>** war«.

Hier der komplette Hauptteil des Programms mit einigen Änderungen gegenüber früher besprochenen Bausteinen. Wir wollen ja schließlich ständig etwas Neues erfahren. Wundern Sie sich also bitte noch nicht, wenn Sie nicht sofort alles wiedererkennen.

Lauffähig ist der folgende Teil natürlich noch nicht, weil der gesamte Deklarierungsteil fehlt. Und da gibt es einiges zu deklarieren!

```
      PROCEDURE (* P28 *) menu3;
      LABEL start;
      BEGIN
(*A*)   start:
(*B*)   ClrScr;
        zuweisung;
        ausdruck;
(*C*)   (* Beginn des Menüs *)
        auswahl:= ['L','R','O','U','V','H','E','A','Z'];
        taste:='X';
(*D*)   WHILE taste <> 'E' DO
        BEGIN (* Beginn der Menüschleife *)
          Read(KBD, taste);
(*E*)     IF UpCase(taste) In auswahl THEN
          BEGIN
            IF Ord(taste) < 96 THEN wh:=3
            ELSE wh:=1;
(*F*)       CASE UpCase(taste) OF
              'O': BEGIN con:=1; oben_unten(wh,con); END;
              'U': oben_unten(wh,2);
              'L': rechts_links(wh,1);
              'R': rechts_links(wh,2);
              'V': vorn_hinten(wh,1);
              'H': vorn_hinten(wh,2);
(*G*)         'E': Exit;
              'A': Goto start;
(*H*)         'Z': zufall;
(*I*)       END; (* Ende von CASE OF *)
          END;
        END; (* Ende des Menüs *)
      END. (* Ende des Hauptprogramms *)
```

Erläuterungen:

(A) Nein, bei dieser Zeile handelt es sich nicht um einen Tippfehler. »start« ist aber dennoch weder Variable noch Konstante, noch der Aufruf einer selbstdefinierten

Funktion oder Prozedur, sondern ein sogenanntes Label. Darunter versteht man eine Markierung im Programm zum Auffinden einer bestimmten Stelle.

Solche Labels können Sie beliebig viele in Ihrem Programm erklären, und zwar ebenfalls im Deklarierungsteil, eben unter der Überschrift **LABEL**. Turbo erkennt solche Definitionen im Programm wieder, wenn man hinter den Labelbezeichner einen Doppelpunkt setzt.

Wir markieren mit »start« ganz einfach den Beginn des Programms. Wenn Sie wollen, können Sie solche Labels auch nach links absetzen, um markierte Stellen besonders schnell wiederzufinden. Aber das ist Formsache, über die wir hier nicht streiten wollen.

(B) Diesen Teil kennen wir schon: Es ist das Herrichten unseres Würfels mit nachfolgender Darstellung auf dem Bildschirm.

(C) Zur Initialisierung (Herstellung des Ausgangszustandes) belegen wir die Variable »taste« mit 'X'; denn wenn wir später das Programm wie vorgesehen mit **<E>** beenden wollen, könnten wir es nie mehr starten, ohne den Rechner vorher ganz abzuschalten, weil Turbo sich auch nach einem Neustart den Inhalt der Variablen »taste« gemerkt hat, und wenn dort ein 'E' zu finden ist, läuft das Programm sofort ans Ende. Also muß in »taste« zunächst mal ein Wert enthalten sein, der nachher gar nicht vorkommt.

(D) Mit **WHILE** taste<>'E' **DO** ... beginnt die Menüschleife. Da nicht nur eine einzelne Anweisung abgearbeitet werden soll, sondern ein ganzer Block, ist dieser zwischen die Begrenzer **BEGIN** und **END** einzuschließen. Egal, was nun dort geschieht, der Rechner kehrt schließlich immer wieder zu **WHILE** zurück und überprüft die Bedingung, unter der er weiterarbeiten soll.

Ist die **WHILE**-Bedingung nicht erfüllt, macht er hinter dem Anweisungsblock, den er sonst zu bearbeiten hätte, weiter.

(E) Das ist die zur Genüge besprochene Tastaturabfrage. Wenn eine der vorgesehenen Tasten gedrückt wurde, kann der Kern des Menüs seinen Lauf nehmen: Je nach Taste wird nun eine Prozedur aufgerufen, die selbstverständlich auch dem entsprechen muß, was man eingegeben hat.

Moment, sagt da vielleicht mancher, das ist doch gar kein Menü; denn auf dem Bildschirm erscheint keine Anzeige, was man überhaupt drücken soll. Letzteres ist richtig, ersteres nicht, denn wir haben zwar eine Auswahl zur Verfü-

gung, arbeiten aber der Einfachheit halber mit einer verdeckten Speisekarte (Auswahl- oder Menütafel).

Es ist natürlich eine Zumutung, einem Nichteingeweihten die Menütafel vorzuenthalten, weil er erst einmal alle möglichen Tasten ausprobieren muß und anschließend deren Wirkung analysieren darf. Das läßt sich aber schnell ändern, indem wir eine oder zwei Zeilen auf dem Schirm reservieren und dort ständig angeben, welche Tastatureingaben welche Folgen haben.

In unserem Beispiel müßte das etwa so aussehen:

```
Drehungen: mit <SHIFT>=links   ohne <SHIFT>=rechts
<L>=links <R>=rechts <O>=oben <U>=unten <V>=vorn <H>=hinten
```

Bei umfangreichen Anweisungen blendet man eine Tafel als Hilfestellung ein, die die notwendigen Erklärungen enthält. Aber das ist hier nicht notwendig.

(F) **CASE..OF** ist eine Turbo-Entscheidungsanweisung, die soviel bedeutet wie »falls diese Größe (Entscheidungsvariable) folgenden Wert hat, dann ist folgendes auszuführen:...«.

Es können nun mehrere Abfragen erfolgen, die im Falle ihres Auftretens in der Entscheidungsvariablen zu den verschiedenen Anweisungen führen.

Ist eine »Drehtaste« gefunden worden, erfolgt der Aufruf der entsprechenden Prozedur, die wir allerdings mit Hilfe von zwei Parametern etwas abgeändert haben. Keine Angst, dies wird sofort im nächsten Abschnitt behandelt.

(G) Noch eine neue Anweisung haben wir aufgenommen: **Exit**. Das bedeutet nichts anderes, als daß das Programm, in welchem dieser Bezeichner steht, verlassen werden soll. Bei uns wird damit das Hauptprogramm verlassen, deshalb ist es damit auch gleichzeitig beendet.

Wird **Exit** in einem Unterprogramm aufgerufen, dann erfolgt der Aussprung in das aufrufende, übergeordnete Programm (Prozedur, Funktion), und zwar hinter den entsprechenden Bezeichner.

(H) **Goto** entspricht dem BASIC-Befehl gleichen Namens. Da wir aber in Turbo keine Zeilennummern verwenden, müssen wir andere Sprungziele beschreiben. Und das sind die oben erklärten Labels.

Wird also die Taste **<A>** gedrückt, dann startet das Programm von neuem u[illegible] zeigt den Würfel in Ausgangsstellung.

(I) Bleibt nur noch das ominöse 'Z', das eine Prozedur mit Namen »zufall« aufruft. Sie ist selbst definiert und wird später abgehandelt, wenn wir die Veränderungen der bereits bekannten Drehprozeduren im Griff haben.

Zur Form:

- **CASE** entscheidungsvariable **OF** stellt mit seinen Entscheidungsverzweigungen praktisch einen eigenen Block dar, der mit einem **END** abgeschlossen werden muß.

11.5 Konzentriert programmieren: 4 Prozeduren in einer P29, P30, P31: Prozedur, Parameter

Schauen wir uns noch einmal die Prozedur an, mit der wir die rechte Ebene des Würfels gegen den Uhrzeigersinn verdrehen können, dann fällt uns – wie schon kurz erwähnt – auf, daß sich im dreidimensionalen **Array** »wuerfel« immer nur die beiden ersten Dimensionen (Schicht und Reihe) ändern, während die letzte immer den Wert 2 behält. Anders ausgedrückt: Nur die Elemente auf Platz zwei werden in ihrer Lage verändert.

Für die Drehung der linken Ebene gilt das eben Gesagte auch, aber eben für die Elemente mit der Platznummer 1. Daraus folgt, daß wir beide Routinen zu einer einzigen zusammenfassen können, wenn wir das letzte Glied im Feld »wuerfel« variabel halten und statt einer festen Belegung mit 1 oder 2 die Variable »con« (von Constante) einsetzen.

Die Prozedur »rechts_links« nimmt dann folgende Form an:

```
(*A*) PROCEDURE (* P29 *) rechts_links(wh,con:Integer);
      (* Drehung von Schicht rechts oder links *)
      VAR z:Integer;
      BEGIN
(*B*)    FOR z:=1 TO wh DO
        BEGIN
(*C*)     retten:=wuerfel[1,1,con];
          wuerfel[1,1,con]:=wuerfel[2,1,con];
          wuerfel[2,1,con]:=wuerfel[2,2,con];
          wuerfel[2,2,con]:=wuerfel[1,2,con];
```

```
          wuerfel[1,2,con]:=retten;
        END;
(*D*)   ausdruck;
      END;
```

Erläuterungen:

(A) Hinter dem Prozedurbezeichner »rechts_links« finden Sie noch zwei Festwertparameter, nämlich »wh« (für Wiederholung) und das eben genannte »con« für eine konstante Zahl. Beide sollen Integerzahlen darstellen.

Blättern Sie noch mal zurück zum vorherigen Abschnitt, in dem wir das Hauptprogramm mit seinem Menü beschrieben haben. Dort finden Sie die Erklärung: Wenn die Eingabe mit der **<SHIFT>**-Taste erfolgte, dann wird »wh« mit 3 belegt, andernfalls mit dem Wert 1.

Die Prozedur »rechts_links« übernimmt diesen Wert im gleichnamigen Parameter »wh«. Wofür er gebraucht wird, sehen wir im nächsten Teil (B).

(B) Eine **FOR**-Schleife zählt nun bis 1 oder bis 3, je nachdem, wie der Inhalt von »wh« aussieht. Und ebensooft wird nun die Verdrehung durchgeführt.

Warum gerade dreimal? Weil eine Drehung gegen den Uhrzeigersinn durch genau drei Drehungen mit dem Uhrzeigersinn ersetzt werden kann. Es wäre also reine Platzverschwendung, wenn wir nochmals eine Extra-Prozedur für den anderen Umlaufsinn schreiben würden. Turbo arbeitet so schnell, daß es niemandem auffällt, daß wir bei Eingabe einer Linksdrehung eigentlich gar keine solche durchführen, sondern die gleichen Stellungen mit drei aufeinanderfolgenden Rechtsdrehungen erzeugen.

Damit haben wir mit einer Prozedur zwei elementare Prozeduren ersetzt.

(C) Wieder ein Blick ins Programm für das Menü: Bei Taste **<L>** wird »con« mit 1 belegt, bei Taste **<R>** mit 2.

Wieder übernimmt ein gleichnamiger Parameter in der Prozedur »rechts_links« eine dieser beiden Möglichkeiten. Sie können es ja einzeln ausprobieren, wenn Sie noch skeptisch sind. Die Vertauschungen der linken bzw. der rechten Ebene lauten:

```
{ Rechts: bei con:=2; }
retten:= wuerfel[1,1,2];
wuerfel[1,1,2]:=wuerfel[2,1,2];
wuerfel[2,1,2]:=wuerfel[2,2,2];
wuerfel[2,2,2]:=wuerfel[1,2,2];
wuerfel[1,2,2]:=retten;

{ Links: bei con:=1; }
wuerfel[1,1,1]:=wuerfel[2,1,1];
wuerfel[2,1,1]:=wuerfel[2,2,1];
wuerfel[2,2,1]:=wuerfel[1,2,1];
wuerfel[1,2,1]:=retten;
```

Damit haben wir also tatsächlich mit einer einzigen, ein klein wenig umfangreicheren Prozedur vier gleichartige auf einmal ersetzt:

1. Drehung der rechten Ebene nach rechts (wh:=1; con:=2)
2. Drehung der rechten Ebene nach links (wh:=3; con:=1)
3. Drehung der linken Ebene nach rechts (wh:=1; con:=2)
4. Drehung der linken Ebene nach links (wh:=3; con:=1)

(D) Anschließend wird die neue Lage sofort angezeigt. Und zwar nur die gewünschte Endlage. Bei einer dreifachen Drehung zeigen wir auch bloß das Endergebnis, die Zwischenformen unterschlagen wir.

Damit Sie nicht so lange herumprobieren müssen, geben wir Ihnen auch noch die beiden anderen Prozeduren an. Sie können damit nun das komplette Verdrehspiel laufen lassen. Ein weiterer Kommentar erübrigt sich. Er würde dem gerade Erklärten entsprechen.

```
(*A*) PROCEDURE (* P30 *) oben_unten(wh,con:Integer);
      (* Drehung der oberen oder unteren Schicht *)
      VAR z:Integer;
      BEGIN
(*B*)   FOR z:=1 TO wh DO
        BEGIN
(*C*)     retten:=wuerfel[con,1,1];
          wuerfel[con,1,1]:=wuerfel[con,1,2];
          wuerfel[con,1,2]:=wuerfel[con,2,2];
          wuerfel[con,2,2]:=wuerfel[con,2,1];
          wuerfel[con,2,1]:=retten;
        END;
(*D*)   ausdruck;
      END;
```

```
(*A*) PROCEDURE (* P31 *) vorn_hinten(wh,con:Integer);
      (* Drehung der oberen oder unteren Schicht *)
      VAR z:Integer;
      BEGIN
(*B*)   FOR z:=1 TO wh DO
        BEGIN
(*C*)     retten:=wuerfel[1,con,1];
          wuerfel[1,con,1]:=wuerfel[2,con,1];
          wuerfel[2,con,1]:=wuerfel[2,con,2];
          wuerfel[2,con,2]:=wuerfel[1,con,2];
          wuerfel[1,con,2]:=retten;
        END;
(*D*)   ausdruck;
      END;
```

Damit dürfte nun der erste Teil des Menüs seine vollständige Erklärung gefunden haben.

11.6 Überlassen wir mal etwas dem Zufall Prozedur P32: Randomize, Random(xx), Random, Delete, Copy

Jetzt fehlt nur noch die Prozedur mit dem Namen »zufall«, die als letzter möglicher Aufruf im Menü steht. Sie wird über die Tasten **<Z>** mit oder ohne **<SHIFT>** aktiviert.

Wir wollen nämlich damit einem Mitspieler einen völlig ungeordneten, nur vom Zufall bestimmten Würfel vorsetzen, den man dann mit unseren Drehmöglichkeiten wieder in die geforderte Ursprungskonfiguration versetzen kann.

Dazu bedienen wir uns Turbo-eigener Prozeduren und Funktionen, die uns Zufallszahlen liefern können. Übrigens ist dies nichts Pascal-Spezifisches, denn Turbo zapft vielfach – so auch in diesem Fall – die betriebsinternen Routinen des Rechners an.

Die Turbo-Prozedur **Randomize**, die ohne jeglichen Parameter aufgerufen wird, startet den eingebauten Zufallsgenerator mit einer Zufallszahl. Von dieser aus werden durch ständige Veränderungen neue Zahlenmuster entworfen. In dem Moment, wo man die Funktion **Random** oder **Random**(xx) aufruft, wird die gerade vorhandene Zufallszahl übergeben.

Dabei gibt es mit **Random** die Möglichkeit, daß man eine Zufallszahl zwischen 0 und 1 erhält, also einen Bruchteil von 1. Der Grenzwert 0 kann auftreten, der Grenzwert 1 nicht.

Oder aber man setzt hinter **Random** eine Klammer und gibt eine Integerzahl an, z.B. 8, dann erhält man in diesem Fall mit **Random**(8) eine Integerzahl zwischen 0 und 8. Der Grenzwert 0 ist möglich, der Grenzwert 8 nicht.

Da **Random** und **Random**(XX) Funktionen sind, können sie nicht einfach mit ihrem Namen aufgerufen werden, sondern nur mit Hilfe einer Zuweisung.

Beispiel: `x:=Random(8);`

Die Variable »x« enthält danach einen ganzzahligen Wert kleiner als 8, aber größer oder gleich 0. Damit sind wir schon dicht vor unserem Ziel: Wir wollen nämlich jedem Miniwürfel unseres Feldes »wuerfel« eine Zahl von 1 bis 8 zuweisen, und zwar bunt gemischt, was uns ja die Funktion **Random**(xx) ermöglichen sollte.

Aber ganz so einfach ist das gar nicht, wie es zunächst aussieht. Tückischerweise können wir nämlich einer einmal zufällig ermittelten Zahl zwar jederzeit eine Position in unserem Zauberwürfel zuweisen. Aber: Wenn wir weiter mit **Random**(8) arbeiten, dann kann es ja sein, daß eine bereits verwendete Zahl noch einmal per Zufall erscheint. Dann würde zwei Miniwürfeln die gleiche Zahl zugewiesen, und das darf nicht sein.

Lassen wir uns also dazu etwas einfallen:

Auf jeden Fall stellen wir sicher, daß alle Elemente unseres dreidimensionalen Feldes »wuerfel« abgeklappert werden, so daß eine Zuweisung sichergestellt ist. Das kennen wir schon, eine dreifach geschachtelte **FOR**-Schleife besorgt das problemlos. Wir müssen eigentlich nur dafür sorgen, daß eine einmal benützte Zahl aus dem Bereich von 1 bis 8 nie mehr auftreten kann.

Und jetzt lohnt es sich, daß wir uns bei den Elementen, den »minis«, für Strings und nicht für Zahlen entschieden haben. Denn eigentlich weisen wir ja keine Nummer als Zahl zu, sondern als Zeichen. (Wenn Ihnen das nicht mehr ganz klar ist, dann schauen Sie noch einmal nach, wie der Ausgangswürfel mit »fixpos« erstellt wurde.) Das erlaubt uns nun, eine Zeichenkette zu definieren, die aus den Zeichen besteht, die wir zuweisen wollen, also aus den Ziffern 1 bis 8. Wir legen sie in der Variablen »kette« an.

Dann lassen wir den Zufall spielen, und der erwischt nun eine ganze Zahl zunächst zwischen 1 und 8. Diese kann nun dem ersten Miniwürfel zugewiesen werden. Um einer unnötigen Herumprobiererei von vorneherein zu begegnen, wird die zuzuweisende Ziffer dem String »kette« entnommen, aber im gleichen Durchgang auch entfernt, so daß »kette« nun um ein Zeichen kürzer ist.

Wenn wir im nächsten Durchgang noch dafür sorgen, daß nun nicht mehr aus dem Bereich von 1 bis 8 die Zufallszahlen entnommen werden, sondern nur noch aus 1 bis 7, sind wir schon einen Schritt weiter. Lassen Sie sich an dieser Stelle nicht täuschen: Die Ziffer 8 ist z.B. durchaus noch machbar, auch wenn sie in den ersten Durchläufen nicht gezogen wurde und »kette« z.B. nur noch 5 Zeichen lang ist. Andererseits macht es überhaupt nichts aus, wenn mehrfach hintereinander die gleiche Zufallszahl auftritt. Aus dem String »kette« werden einfach beliebige Stellen abgebaut.

Gehen wir mal ein paar Schritte an einem frei ausgedachten Fall durch:

1. Durchlauf

Ausgangssituation: kette='12345678'
Random(8)+1 liefert den Wert 5.
Zeichen Nr.5 aus »kette« ist '5'.
Miniwürfel [1,1,1] erhält die Ziffer '5'.
Das fünfte Zeichen wird aus »kette« entfernt: kette='1234678'

2. Durchlauf

Ausgangssituation: kette='1234678'
Random(7)+1 liefert den Wert 5.
Zeichen Nr.5 aus »kette« ist '6'.
Miniwürfel [1,1,2] erhält die Ziffer '6'.
Das fünfte Zeichen wird aus »kette« entfernt: kette='123478'

usw.

Haben Sie den Trick erkannt? Er liegt darin, daß man im nächsten Durchgang jeweils den Bereich der Zufallszahlen um 1 erniedrigen muß.

Warum man immer noch 1 zu der gewonnenen Zufallszahl addieren muß? Weil man z.B. mit **Random**(8) niemals die Zahl 8 erreichen kann, wohl aber mit **Random**(8)+1.

Warum man dann nicht gleich **Random**(9) wählt? Weil man damit statt der vor-

gesehenen 8 Möglichkeiten deren 9 programmiert; denn die Null ist ja auch möglich (siehe oben).

Wie man das Ganze per Programm realisiert? Hier ist die Lösung:

```
      PROCEDURE (* P32 *) zufall;
(*A*) VAR
        zu,x:Integer;
        ziff:st;
        kette:String[8];
      BEGIN
        X:=0;
        kette:='12345678';
(*B*)   Randomize;
(*C*)   FOR si:=1 TO 2 DO
        FOR ri:=1 TO 2 DO
        FOR mi:=1 TO 2 DO
        BEGIN
          zu:=Random(8-x)+1;
(*D*)     wuerfel[si,ri,mi]:=Copy(kette,zu,1);
(*E*)     Delete(kette,zu,1);
(*F*)     x:=x+1;
        END;
        ausdruck;
      END;
```

Erläuterungen:

(A) Es ist zweckmäßig, die Variablen, die nur in der Prozedur benötigt werden, auch dort zu definieren, denn Turbo verwaltet sie für solche Blöcke extra und braucht deshalb nicht erst im Verzeichnis der globalen Variablen nachzusehen.

»x« wird benötigt, um den Zahlenbereich, aus dem wir die Zufallszahlen ziehen lassen, jeweils um 1 zu vermindern, bevor der nächste Durchgang beginnt. »zu« nimmt die Zufallszahl aus der **Random**-Funktion auf.

Zur Initialisierung (Anfangszustand herstellen) wird der Reduzierer »x« auf Null gesetzt, so daß der erste Durchlauf mit dem größtmöglichen Zahlenbereich für die Auswahl der Zufallszahl beginnen kann. »kette« enthält anfangs alle 8 Ziffern.

(B) Starten des Zufallsgenerators mit einer zufälligen Anfangszahl.

(C) Die geschachtelte **FOR**-Schleife weist insgesamt 8 Elementen des **Array** »wuerfel« (2 mal 2 mal 2) die Ziffern zu.

(D) Das ist die eigentliche Zuweisung. Dem String »kette« wird mit **Copy** jeweils ein Zeichen beginnend mit »zu« entnommen. Das ist also immer genau das an der Stelle »zu« befindliche Zeichen.

Nach dem **Copy**-Befehl bleibt »kette« unverändert. Entnommen heißt in diesem Fall noch lange nicht entfernt.

(E) Das Entfernen eines Zeichens aus »kette« nehmen wir mit **Delete** vor. Diese Standardprozedur kürzt einen ursprünglichen String von einer bestimmten »position« ab um eine bestimmte »anzahl« Zeichen. Der Rest wird zusammengeschoben zu einem neuen String, der den alten Namen (hier »kette«) beibehält. Der ursprüngliche längere String geht damit verloren.

Schreibweise: `Delete(zeichenkette,position,löschanzahl)`

In unserem Programm wird immer nur ein Zeichen entfernt.

(F) Vor dem Start der nächsten Schleife wird das Korrekturglied »x« um 1 erhöht, um bei **Random**(8-x)+1 einen um eine Einheit kleineren Bereich für die Zufallszahlen zu erzeugen.

Sie sollten nun in der Lage sein, sich das vorläufig komplette Programm aus den gebotenen Bausteinen zusammenzustellen. Zur Hilfestellung geben wir nochmals den Aufbau an:

Teil 1: Globaler Deklarierungsteil mit

```
TYPE ...
VAR ...
CONST ...
LABEL ...

PROCEDURE ausdruck;
PROCEDURE rechts_links(wh,con:Integer);
PROCEDURE oben_unten(wh,con:Integer);
PROCEDURE vorn_hinten(wh,con:Integer);
PROCEDURE zuweisung;
PROCEDURE zufall;
```

Teil 2: Anweisungsteil (Hauptteil) mit

```
Initialisierungen
Menüschleife
```

Das Listing des lauffähigen Testprogramms »P22/P23.DEM« drucken wir hier ab. Vor dem Ausprobieren müssen Sie die bereits besprochenen Prozeduren als Include-Dateien auf Diskette bereithalten.

```
(*************************************************************)
(*          TURBO PASCAL auf dem Schneider CPC 6128          *)
(*             Demo zum Testen des Drehwuerfels              *)
(*                 Markt & Technik MT 90455                  *)
(*                   (C) W. Kassera 1987                     *)
(*************************************************************)

     PROGRAM P22_P23;  (*dreidimensional *)
     TYPE
       st=String[1]);
       mini=st;
       reihe=Array [1..2] Of mini;
       schicht=Array [1..2] Of reihe;
       wuerfel3=Array [1..2] Of schicht;
     VAR
       wuerfel:wuerfel3;
       si,ri,mi,ze,sp,wh,con:Integer;
       nummer,retten:st;
       auswahl:Set Of Char;
       taste:Char;
     CONST
      fixpos:wuerfel3=((('1','2'),('3','4')),(('5','6'),('7','8')));
     {$I AUSDRUCK.P23}
     {$I P29}
     {$I P30}
     {$I P31}
     {$I P32}
     {$I ZUWEISEN.P22}
     PROCEDURE menu;
     Label start;
     BEGIN
       start:
       ClrScr;
       zuweisung;
       ausdruck;
       Gotoxy(1,1);Write('Taste  ( L,R,O,U,V,H,E,A,Z ) ? ');
```

```
  (* Beginn des Menüs *)
  auswahl:= ['L','R','O','V','H','E','A','Z'];
  taste:='X';
  WHILE taste <> 'E' DO
  BEGIN  (* Beginn der Menüschleife *)
    Gotoxy(32,1);
    Read (KBD,taste);
    IF UpCase(taste)  In auswahl THEN
    BEGIN
      IF Ord(taste) < 96 THEN wh:=3
      ELSE wh:=1;
      CASE UpCase(taste) OF
        'O': BEGIN con:=1; oben_unten(wh,con); END;
        'U': oben_unten(wh,2);
        'L': rechts_links(wh,1);
        'R': rechts_links(wh,2);
        'V': vorn_hinten(wh,1);
        'H': vorn_hinten(wh,2);
        'E': Exit;
        'A': Goto start;
        'Z': zufall;
      END;     (* Ende von CASE OF *)
    END;       (* Ende der Menüschleife *)
  END;         (* Ende der Prozedur Menü *)
END;
BEGIN (* Hauptprogramm *)
  menu;
END.
```

Kapitel

12

Zeit ohne Uhr: Arbeiten mit Zeigervariablen

Bevor wir das nächste Programm realisieren, das eine Zeitbestimmung ohne Uhr durchführen soll, gehen wir auf eine Besonderheit von Turbo ein, die in anderen Sprachen wie z.B. in BASIC nicht zu finden ist. Es sind Variablen, die man je nach Bedarf plazieren oder entfernen kann.

Nachdem sie in der Regel recht stiefmütterlich behandelt werden, weil sie etwas aus dem Rahmen des üblichen fallen, gehen wir an Hand eines »Uhrenbeispiels« etwas näher auf sie ein.

Gerade sie sind es, die Pascal lebendig machen. Sie sollen deshalb nicht nur »der Vollständigkeit halber erwähnt werden.«

Sie sollten die folgenden Abschnitte nicht überschlagen, sondern sich die Vorüberlegungen zu Gemüte führen, um das eigentliche Programm besser verstehen zu können.

12.1 Ein weiterer Datentyp: der Zeiger

Pascal kennt neben den üblichen Variablen, die mit Hilfe eines Variablennamens aufgerufen werden, auch noch eine ganz andere Art: Es sind sogenannte Zeiger- oder Pointertypen.

Normalerweise werden in einem bestimmten Speicherbereich die Variablen in der Reihenfolge angelegt, wie sie anfallen. Entsprechend ihrem Platzbedarf dehnen sie sich von einer niedrigen Adresse in die höheren aus. Während des Programmlaufs können und dürfen sie ihre Position nicht verändern. Sie bleiben an Ort und Stelle, man sagt, sie sind statische Variablen.

Im Gegensatz dazu stehen die sogenannten dynamischen Variablen, die je nach Bedarf angelegt, aber auch total entfernt werden können und damit den Platz wieder freigeben für andere Variablen.

Solche dynamischen Variablen können natürlich nicht zusammen mit den statischen in einem gemeinsamen Speicherbereich untergebracht werden. Sie würden sich gegenseitig stören und die strenge Pascal-Ordnung durcheinanderbringen. Deswegen wurde extra für sie ein eigener Adreßraum bereitgestellt, der sich Heap nennt. Seine Lage ist je nach Betriebssystem unterschiedlich, doch in jedem Fall flexibel. Das heißt, seine Unter- und Obergrenze ist dem Platzbedarf angepaßt, den die übrigen Teile des Programms benötigen.

Mehr an grauer Theorie ist im Moment gar nicht notwendig. Es genügt uns zu wissen, daß es möglich ist, diesen Heap ähnlich wie einen Stapel (Stack) zu nutzen und mit Inhalten zu füllen.

Dazu eben benötigt man die sogenannten Zeigervariablen. Sie werden deklariert wie andere Variablen auch, weisen aber den entscheidenden Unterschied auf, daß sie vom Typ »Zeiger« sind, was mit dem Kontrollzeichen »^« ausgewiesen wird.

Das kann dann so aussehen:

```
TYPE
  zeigertyp=^Integer;
VAR
  z1,z2,z3: zeigertyp;
```

Das bedeutet, daß alle Variablen vom Typ »zeigertyp« nicht bei den gewöhnlichen Variablen angelegt werden sollen, sondern daß Integerzahlen erwartet werden, die sich

im Heap wiederfinden lassen. Selbstverständlich lassen sich auch alle anderen Datentypen wie **Real**, **String**, **Array** usw. mit Hilfe von Zeigertypen verwalten. Wir werden das im nächsten Beispiel sehen.

Eine weitere grundsätzliche Eigenschaft der Zeigervariablen »z1«, »z2« und »z3« ist aber die, daß diese Bezeichner gar keine Namen verkörpern, sondern Zweibyte-Adressen, eben sogenannte Zeiger. »z1« sagt also später im Programm nur aus, wo der Inhalt dieser Variablen zu finden ist. Deswegen werden die Zeigervariablen auch manchmal anonyme (namenlose) Variablen genannt.

Wir sollten uns zumindest folgendes merken: Der Bezeichner einer Zeigervariablen steht immer stellvertretend für eine Adresse im Heap.

12.2 Platz für die Zeigervariablen: Heap, New, GetMem

Die Adresse einer Zeigervariablen muß irgendwann einmal festgelegt werden. Das kann aber erst beim Programmlauf der Fall sein, denn bis jetzt ist in unserem Beispiel noch gar nichts geschehen, wir haben lediglich einen Datentyp und drei Variablen deklariert. Wo diese drei Zeigervariablen dann tatsächlich eingerichtet werden, hängt davon ab, wie wir sie im Anweisungsteil vorbehandeln. Denn im Gegensatz zu den statischen Variablen ist ihnen ja bisher noch kein Platz zugewiesen worden.

Dafür stehen zweierlei Anweisungen zur Verfügung:

Die erste Möglichkeit ist **New**, was nichts anderes bedeutet, als daß eine neue Zeigervariable eingestellt werden soll.

Passend zu unserem Beispiel könnte es dann im Anweisungsteil lauten:

```
New(z1),New(z2),New(z3)
```

In diesem Fall weiß Turbo selbst genau, wieviel Platz es für jede dieser Variablen vorsehen muß, nämlich genau vier Byte, von denen zwei für die Integerzahl reserviert werden, während die nächsten beiden von Turbo mit 0 belegt werden. (Warum, wissen wir nicht.)

Auf dem Heap sieht es nun folgendermaßen aus:

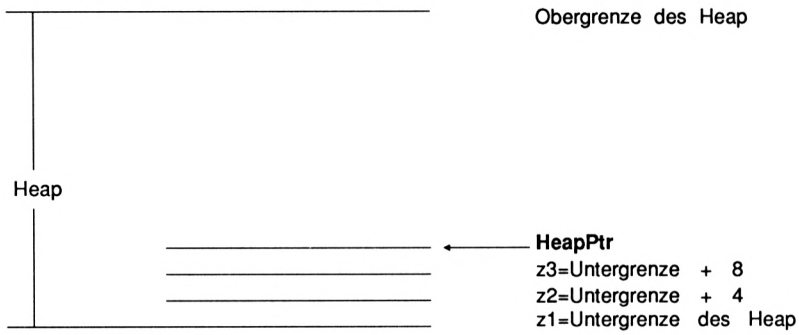


Bild 12-1: Belegung des Heap

Dabei treffen wir wieder auf eine vordeklarierte Variable, nämlich den sogenannten Heap-Zeiger **HeapPtr**, der uns beim Rekursionsproblem schon begegnet ist. Mit ihm wird sozusagen die Spitze des Heap angezeigt. Auch er ist vom Typ Zeiger.

Die zweite Möglichkeit, einer Zeigervariablen ihren Platz zuzuweisen, sieht so aus:

```
GetMem(z1,4);
GetMem(z2,6);
GetMem(z3,8);
```

Der Unterschied zu **New** liegt darin, daß wir nun selbst bestimmt haben, wie viele Bytes Platz wir den einzelnen Variablen einräumen. In unserem Beispiel ist es zwar nicht sehr sinnvoll, Integerzahlen mehr als zwei Bytes zuzugestehen, aber wir wollen damit gleich etwas anderes ausprobieren.

12.3 Entfernen unnötiger Variablen

Sollten Zeigervariablen nicht mehr benötigt werden, genauer gesagt, sollte der Platz, den ihre Inhalte einnehmen, wieder freigegeben werden, dann bietet Turbo dafür gleich drei verschiedene Versionen an, die sich in ihrer Wirkung erheblich unterscheiden.

12.3.1 Release gibt Heap-Speicherplatz frei

Weil der Heap wie ein Stapel verwaltet wird, kann man an Hand des Programmverlaufs ständig mitverfolgen, wie sich auf dem Heap die Zeigervariablen auftürmen. In

der Reihenfolge, wie sie mit **New** oder **GetMem** angelegt werden, stehen sie bereit: Die zuletzt eingerichtete Zeigervariable bildet die Heap-Spitze.

Will man nun auf dem Heap Variablen entfernen, um für andere wieder Platz zu schaffen, dann läßt sich mit **Release** und der Angabe eines Zeigers der gesamte Speicherbereich des Heap oberhalb der angegebenen Variablen »abräumen«.

Beispiel: Mit **New** wurden die vier Zeiger Z1 bis Z4 eingerichtet. **Release**(Z3) stellt den Heap-Zeiger auf den Anfang der Zeigervariablen Z3^ und schafft damit ab hier Platz für neue Zeigervariablen.

Nebenbei zeigt sich folgende Erscheinung: Z3 wird sofort unbrauchbar, die darüberliegenden Zeiger aber erst dann, wenn mit **New** oder **GetMem** neue Zeiger angefordert werden.

Anmerkung: Laut Handbuch sollte vor **Release**(Zeiger) noch der Befehl **Mark**(Zeiger) gegeben werden, um die gewünschte Position des Heap-Zeigers zunächst zu markieren. Das funktioniert in unserer CP/M-Version nicht. Im Gegenteil: **Mark**(Zeiger) verhindert ein Freigeben. **Release**(Zeiger) genügt also.

12.3.2 Freigeben eines einzelnen Blocks mit Dispose

Mit **Dispose** wird auf eine einzige Zeigervariable zugegriffen, und diese wird aus dem Heap entfernt.

Wählen wir entsprechend unserem Beispiel

```
Dispose(z2);
```

dann bleibt der Zeiger »z1« erhalten wie bei **Release**, aber auch »z3« ist aktiv. Dabei hat sich der freie Heap-Bereich aber nicht verändert. Lediglich der Raum zwischen »z1« und »z2« wird nicht mehr verwendet. Ein Zusammenschieben der vorhandenen Zeigerbereiche erfolgt nicht.

Es ist daher nicht zweckmäßig, **Dispose** und **Release** in bunter Mischung in einem Programm zu verwenden, da man schnell den Überblick verliert, was auf dem Heap geschieht.

12.3.3 FreeMem schafft Platz

Die dritte Möglichkeit, anonyme Variablen wieder in der Versenkung verschwinden zu lassen, bringt **FreeMem**, das eine genau anzugebende Anzahl von Bytes beginnend bei der Anfangsadresse eines Zeigers freimacht und den Rest wieder zusammenschiebt, so daß der Heap-Zeiger nach **FreeMem** wieder auf einen niedrigeren Wert gesetzt werden kann.

FreeMem kann man natürlich nur dann anwenden, wenn man weiß, wieviel Platz eine Zeigervariable benötigt. Und das wiederum kann man ohne genauere Kenntnisse der Typenverwaltung eigentlich nur dann, wenn man zuvor mit **GetMem** einen bestimmten Bereich zugewiesen hat.

Beispiel: Nehmen wir an, wir haben mit **GetMem** entsprechend dem Beispiel aus 12.1 die Zeiger »z1« bis »z3« angelegt. Dann können wir z.B. »z2« wie folgt entfernen:

```
FreeMem(z2,6);
```

»z1« und »z3« sind nach wie vor aktiv. Aber »z3« hat nun eine andere Anfangsadresse.

Was passiert, wenn man **FreeMem** unkontrolliert anwendet, also ohne genau die Anzahl Bytes anzugeben, die man vorher bestellt hatte, kann man sich leicht ausmalen. Es gibt Überschneidungen in den Zeigervariablen, die sofort zum Datenchaos führen.

12.4 Überprüfungen und Zuweisungen Programm P33: MemAvail, Addr, Ptr

Will man wissen, wieviel freier Platz (noch) auf dem Heap zur Verfügung steht, dann wendet man die Standardfunktion **MemAvail** an. Wer den tatsächlichen Wert der Zeigeradresse ermitteln will, braucht dazu nur die Standardfunktion **Addr**(Variablenname) aufzurufen.

Beispiel: `Write(Addr(z1^));`

Die Ausgabe erfolgt in einer (Integer-)Dezimalzahl. Sollte eine negative Ausgabe erscheinen, dann ist die reelle Zahl 65636.0 zu addieren, da beim Überschreiten des Integerbereichs von 32768 höhere Werte als Komplement zu 65536 erzeugt werden.

Mit der Funktion **Ptr** dagegen bewirkt man das Gegenteil: Einer bereits deklarierten Zeigervariablen kann eine ganz bestimmte Adresse zugeordnet werden.

Für die 8-Bit-Systeme genügt als Argument eine Integerzahl wie z.B.

```
z1:=PTR($D000);
```

In unserem Beispiel würde der Zeiger »z1« auf die Adresse 53248 gestellt, so daß eine dort beheimatete Integerzahl die Adressen 53248 und 53249 beanspruchen würde.

Zur Form:

- Beachten Sie bitte die Schreibweisen: Wenn der Zeiger als Adresse angegeben werden soll, steht zu seiner Beschreibung nur der Bezeichner zur Verfügung: z.B. **GetMem**(z1,2) oder **New**(z1) legen jeweils die Anfangsadresse des Zeigers »z1« fest.

- Ansonsten muß bei Zuweisungen oder ähnlichem immer das Zeichen »^« mit angegeben werden, damit Turbo Zeigervariablen-Inhalte von Variablenadressen unterscheiden kann: z.B. Zeigeranfang:=**Addr**(z1^) holt die Adresse von der Stelle, an der der Variableninhalt beginnt.

  Aber z1^:=$D000 belegt die beiden Bytes an der Zeigeradresse mit Lo=0 und Hi=$D0.

- Um mit der Zahl 65536 operieren zu können, ist sie als Realzahl in der Form 65536.0 anzugeben, weil Turbo sonst mit 65536 eine Integerzahl außerhalb des zulässigen Bereichs erkennt und dies mit einer Fehlermeldung ahndet.

- Turbo verarbeitet auch Hexzahlen bis 65535, wenn man sie in der Form $XX verwendet, also das Dollarzeichen voranstellt.

Fassen wir unsere bisherigen Erkenntnisse über Zeiger und die jeweils zugehörigen Turbo-Anweisungen mit einem etwas ausführlicheren Testprogramm zusammen, das Sie mit etwas Vorsicht durchaus selbst variieren sollten.

```
PROGRAM (* P33 *) Zeigertest;
TYPE
  wert=Integer;
  zeigertyp=^wert;
```

```
      VAR
        z1,z2,z3,z4,z5:zeigertyp;
        i:Integer;
      PROCEDURE max;
      BEGIN
        Write('verfuegbar: ', MaxAvail,'  ');
      END;
      PROCEDURE heapzeiger;
      BEGIN
        WriteLn('Heap-Zeiger: ', HeapPtr);
      END;
      PROCEDURE druck;
      BEGIN
        Write('Inhalt von z1: ',z1^);
        WriteLn('Inhalt von z2: ',z2^);
        max;heapzeiger;
      END;
      PROCEDURE dispo;
      BEGIN
        Dispose(z2);
        WriteLn('nach Dispose(z2): ');
        druck;
      END;
      PROCEDURE neuheap;
      BEGIN
        HeapPtr:=HeapPtr+$0100;
        druck;
      END;
      BEGIN
        ClrScr;
        Write('anfangs: ');max;heapzeiger;
        New(z1);New(z2);
        WriteLn('nach New(z1)..(z2): ');max;heapzeiger;
        z1^:=10000;z2^:=100;
        FOR i:=0 TO 8 DO
        Write(Mem[Addr(z1^)+i],' ');
        druck;
        WriteLn('Nach Mark/Release: ');
        {Mark(z1);}Release(z1);
        druck;
        GetMem(z3,20);
        WriteLn('Nach GetMem(z3,20): ');
        druck;
(*A*)   z4:=z2;z2:=Nil;
        WriteLn('Nach Nil(z2): ');
        druck;
(*B*)   z2:=z4;
        druck;
```

```
    WriteLn('Adresse von z1: ', Addr(z1^));
    WriteLn('Adresse von z2: ', Addr(z2^));
    FreeMem(z3,20);
    druck;
  END.
```

Bei (A) haben wir noch eine neue Zuweisungsmöglichkeit eingefügt: Sie lautet **Nil** und weist dem Zeiger den Wert 0 zu. Damit wird ausgedrückt, daß dieser Zeiger zur Zeit nicht in Betrieb ist und den Wert 0 hat.

Um seine ursprüngliche Lage zu retten, weisen wir ihn deshalb vorher einem anderen Zeiger zu. Braucht man den alten Zeiger mit seinem ursprünglichen Wert wieder, dann läßt er sich wieder herstellen wie in Zeile (B) gezeigt.

Am besten, Sie probieren dieses Testprogramm an Ihrem Gerät aus. Es erklärt sich von selbst. Die »gefährlichen« Operationen haben wir vorläufig mit Kommentarklammern stillgelegt. Wenn Sie bereits Übung haben, dann entfernen Sie sie und schauen sich die Ausdrucke in Ruhe an. Wundern Sie sich aber nicht über sonderbare Reaktionen Ihres Rechners, wenn Sie beispielsweise einer Zeigervariablen einen Wert zuweisen, ohne sie vorher mit **New** oder **GetMem** aktiviert zu haben.

12.5 Zählwerk zur Bestimmung der Zeitdauer Programm P34: Arbeiten mit Zeigern, Rechnen mit Integerzahlen außerhalb des Standardbereichs

Unser Programm zum Experimentieren mit dem kleinen Zauberwürfel ist ja schon fast perfekt. Aber eins fehlt doch noch: Man sollte auch die Zeit bestimmen können, die man braucht, um den Würfel aus einer zufälligen Stellung wieder in Normallage zu bringen.

Dazu messen wir immer den Zeitraum, der zwischen der letzten Verdrehung und dem Beginn der folgenden liegt. Der Start beginnt dann, wenn der Würfel übel zugerichtet auf dem Schirm erscheint. Das Ende der Zeitmessung legen wir mit Drücken der Taste **<E>** fest, die dann bedient werden soll, wenn die ursprüngliche Ausgangssituation wieder vorliegt.

Wie gehen wir nun vor?

Während der Spieler vor dem Schirm sitzt und sich überlegt, ob er nun die obere

Ebene nach links oder lieber erst die rechte nach vorn drehen soll, lassen wir eine Warteschleife laufen, die bei jedem Durchgang einen Zähler um 1 erhöht (inkrementiert).

Erfolgt ein Tastendruck, weil sich der Würfeldreher nun endlich zu einer Entscheidung durchgerungen hat, unterbrechen wir das Zählen, bis das Ergebnis seiner Bemühungen auf dem Schirm erscheint.

Also das Ziel ist klar: Zählen von Null an aufwärts bis zum Auftreten eines Tastendrucks.

Nun hat Turbo die bestechende Eigenschaft, eine Warteschleife mit unwahrscheinlicher Geschwindigkeit zu durchrasen, so daß wir mit einer einzigen Integervariablen nicht allzuweit kommen. Der höchste Wert ist nämlich 32767, dann geht's negativ weiter zurück bis 0, und dann wiederholt sich das ganze Spielchen, weil Integerzahlen eben nur in zwei Bytes (oder einem Wort) abgelegt werden.

Und nun machen wir Nägel mit Köpfen: Wir legen mit Hilfe der eben besprochenen Zeiger drei Integervariablen an, von denen jede einen Wert von maximal 10000 annehmen kann. Dies ist eine willkürliche Grenze, aber mit dieser Zahl läßt sich leicht rechnen.

Wir begeben uns nämlich zunächst vom geliebten Dezimalsystem weg ins Zehntausendersystem:

- In der Zeigervariablen »z1« erfassen wir die Einer. Immer wenn sie bei 10000 überzulaufen droht, setzen wir sie wieder auf Null zurück und erhöhen den Inhalt von »z2«. Dort sitzen also die Zehntausender.

- Wenn auch noch die Zehntausender voll sind, dann sind wir bei 10000 mal 10000, also bei hundert Millionen angelangt. Jetzt erst wird der Inhalt von »z3« um 1 erhöht.

Schematisch sieht unser Zählwerk dann so aus:

100 Millionen	Zehntausender	Einer

Zur Umrechnung in die uns besser bekannte Zeiteinheit »Sekunde« müssen wir aber erst eine Eichung vornehmen. Dazu lassen wir das Zählwerk anlaufen und stoppen bis zum Drücken einer Taste die Zeit mit einer »richtigen« Uhr.

Jetzt wissen wir, wie viele Zähler für eine Sekunde benötigt wurden und können nun jedes beliebige Zählergebnis umrechnen. Dazu bedienen wir uns aber eines Zahlentyps, der etwas mächtiger ist als der Integertyp: Das sind die Realzahlen, die immerhin einen Bereich bis 10 hoch 38 (100 Sixtillionen) abdecken.

```
      PROGRAM (* P34 *) wartezahl;
      TYPE
(*A*)   zeigertyp=^Integer;
      VAR
(*B*)   z1,z2,z3:zeigertyp;
        zahl:Real;
      BEGIN
        ClrScr;
        Write('Uhr ist gestartet. Zum Stop beliebige Tastetippen!');
(*C*)   New(z1);New(z2);New(z3);
        {GetMem(z1,16),GetMem(z2,16); GetMem(z3,16);}
(*D*)   z1^:=0;z2^:=0;z3^:=0;
(*E*)   REPEAT
          z1^:=z1^+1;        {GotoXY(1,5);Write(z1^);}
(*F*)     IF z1^>10000 THEN
          BEGIN
            z1^:=0;
            z2^:=z2^+1;      {GotoXY(1,6);Write(z2^);}
(*G*)       IF z2^>10000 THEN
            BEGIN
              z1^:=0;
              z2^:=0;
              z3^:=z3^+1;
            END;
          END;
(*H*)   UNTIL KeyPressed;
        WriteLn(^J^J^M,'Kontrollzahlen: ');
        WriteLn(z1^);WriteLn(z2^);WriteLn(z3^);
(*I*)   zahl:=(z3^*1.0E8+z2^*1.0E4+z1^);
        {Mark(z1);}Release(z1);
        {FreeMem(z1,16);FreeMem(z2,16);FreeMem(z3,16);}
(*J*)   WriteLn(z3^*1.0E8+z2^*1.0E4+z1^);
        WriteLn(zahl,'   ',zahl/5300:8:2, 'Sekunden');
      END.
```

Erläuterungen:

(A) Mit »^Integer« legen wir fest, daß alle folgenden Daten vom Typ »ZeigerTyp« von einem Zeiger verwaltet werden.

(B) Die Zeiger, mit denen wir die Integerzahlen bearbeiten wollen, heißen »z1«, »z2« und »z3«.

»zahl« ist eine Realzahlvariable, die wir zur Umrechnung benötigen.

(C) Erst hier erfolgt die Zuweisung der Adressen. Wir bieten beide Möglichkeiten an. Bei **GetMem** sind wir großzügig und geben jeder Zeigervariablen einen Raum von einem Paragraphen (16 Byte).

(D) Vor dem erstmaligen Hochzählen werden unsere Einheiten auf Null gesetzt. Denn Turbo vergißt bei einem eventuellen zweiten Start die alten Werte nicht.

(E) Wir wählen eine **REPEAT**-Schleife und erhöhen immer den Integerwert, der durch Zeiger »z1« bestimmt ist.

(F) Falls die Einer den Grenzwert von 10000 überschreiten wollen, werden sie durch Zurücksetzen auf Null daran gehindert. Ein Zehntausender mehr wäre voll.

(G) Falls damit auch weitere 100 Millionen gefüllt sein sollten, werden sowohl die Einer als auch die Zehntausender zurückgesetzt und die 100 Millionen um 1 erhöht.

(H) Die Schleife läuft so lange, bis eine Taste gedrückt wird. In diesem Fall lassen wir uns erst einmal ausgeben, was in den einzelnen Zählern enthalten ist.

(I) Den Wert unseres Zählerstandes erhalten wir, wenn wir die höchstwertige Stelle mit 100000000 multiplizieren, die zweite Stelle mit 10000 und alles zu den noch vorhandenen Einern addieren.

Anschließend geben wir den benützten Heap-Bereich wieder frei. Entsprechend den Zuweisungen sind hier wieder beide Versionen angeboten.

(J) Ein kleiner Test: Wir versuchen jetzt noch einmal, das Ergebnis des Zählers mit Hilfe der drei Zeiger auf den Schirm zu bringen. Es wird uns nicht gelingen. Das Ergebnis ist 0, weil die Zeiger entfernt worden sind.

Rechtzeitig haben wir deshalb den Zählerstand in »zahl« gerettet und können ihn als Sekundenangabe aufzeigen. Der hier auftauchende Faktor 5300 wurde auf dem Schneider CPC 6128 per Stoppuhr ermittelt. Sobald Sie aber die Kontrollausgaben in den Kommentarklammern aktivieren, sinkt die Zählgeschwindigkeit gewaltig. Der »Sekundenfaktor« ist dann neu zu bestimmen.

Zur Form:

- Bei Zuweisungen muß die Zeigervariable stets mit dem Zeichen '^' gekennzeichnet werden. Nicht aber bei direkten Zeigeroperationen, die den Adreßraum betreffen wie bei **FreeMem** oder **Mark/Release**, **GetMem** oder **New**.

- Bei der Auswertung können wir die Integerinhalte der Zeigervariablen durchaus mit einer reellen Zahl multiplizieren. Aber eine Million nimmt Turbo nicht mehr ohne weiteres als siebenstellige Zahl an. Hier arbeiten wir mit Zehnerpotenzen und der entsprechenden Schreibweise:

  Beispiele: 1.0E8 heißt 1 mal 10 hoch 8 (=100000000),
  1.0E4 heißt 1 mal 10 hoch 4 (=10000)

  Versuchen Sie ruhig einmal, hier mit Integerzahlen zu arbeiten. Sie werden sehen, das klappt nicht. Wenn Sie jedoch für eine Million 1000000.0 schreiben, wird dieser Wert als Realzahl akzeptiert.

So, dann wollen wir das Ganze mal starten. Nach dem Kompilieren drücken Sie **<R>**, und gleichzeitig schauen Sie auf Ihre Uhr. Nach einigen Sekunden können Sie eine Taste betätigen.

Vergleichen Sie nun die angegebene Zeit mit der Ihrer Stoppuhr. Wenn Sie sehr danebenliegen, brauchen Sie unser Programm nicht zu beschimpfen. Ändern Sie den Faktor bei der Sekundendarstellung entsprechend ab.

Eines ist klar: Diese Aufgabe mit dem Messen eines Zeitintervalls läßt sich natürlich auch auf sehr viele andere Arten durchführen. Wir können Ihnen ja mal ein paar Variationen zum Ausprobieren bieten:

- Was glauben Sie, geschieht, wenn wir statt des Integerzeigers einen Realzahlzeiger verwenden (zeigertyp=^**Real**;)? Hätte das nicht den Vorteil, daß wir mit einem einzigen Zeiger auskommen, der ja sehr weit hochgezählt werden kann?

- Oder wenn wir überhaupt keine Zeigervariablen, sondern gewöhnliche statische Variablen wählen? Auch hier bieten sich die beiden Möglichkeiten **Integer** oder **Real** an.

Wir können Sie beruhigen: Alle diese Vorschläge sind realisierbar. Die Unterschiede ergeben sich erst bei der Umrechnung in die Einheit Sekunde. Probieren Sie's doch aus!

Eines verraten wir Ihnen im voraus: Das Rechnen mit Integerzahlen ist immer erheblich schneller als mit Realzahlen. Wieviel schneller, das können Sie nun mit dem Programm P34 austesten.

Nachdem wir nun eine Zeitmeßanlage entworfen haben, sollten wir sie auch in unser Würfelspiel einbauen. Dazu verwandeln wir das Programm P34 in eine Prozedur, indem wir den Kopf mit **PROCEDURE** Wartzahl.I34 überschreiben und den Punkt hinter **END** durch einen Strichpunkt ersetzen. Schließlich speichern wir diese Prozedur auf Diskette und verwenden sie als Include-Datei für den Ausbau unseres Würfelspiels.

Der Start eines jeden neuen Spiels muß nun damit beginnen, daß die Zähler z1^ bis z3^ auf Null gesetzt werden. Also werfen wir die Zeile bei (D) hinaus und fügen sie dort ein, wo die »Uhr« mit Null loslaufen soll. Das ist immer dann der Fall, wenn mit der Taste **<Z>** ein neues Spiel beginnt oder wenn mit **<W>** die gleiche Stellung noch einmal durchgespielt werden soll.

Die Prozedur »Wartezahl« rufen wir an der Stelle auf, wo auch tatsächlich gewartet wird. Das ist vor der Tastaturabfrage mit **Read** möglich. Erst wenn ein Tastendruck erfolgt, wird die Warte- und Zählprozedur unterbrochen, bis der Würfel in der neuen Lage erscheint oder bis mit **<E>** das Ende des Spiels angezeigt wird.

Wie immer: Probieren geht über Studieren. Wem unsere Version nicht gefällt, der macht es eben anders. Turbo läßt viele Wege offen.

Kapitel

13

Anwendung von Turbo-System-Funktionen

Wenn Sie das Turbo-Handbuch schon einmal nach Befehlen durchsucht haben, die Ihnen von BASIC her geläufig sind, dann werden Sie sicher auch enttäuscht festgestellt haben, daß es z.B. keine eigenen Befehle gibt, mit denen man den Cursor über den Bildschirm bewegen kann (außer die direkte Ansteuerung einer Bildschirmadresse mit **GotoXY**) oder mit denen man den Bildschirm-Modus wechseln könnte.

Und doch kann man dies alles auch von einem Turbo-Programm aus und noch viel mehr. Denn Turbo bedient sich etlicher Funktionen, auf die man mit einem einfachen **Write**-Befehl ebenfalls zugreifen kann.

Wir haben in mühseliger Kleinarbeit diesen Code geknackt, weil es darüber so gut wie keine Informationen gibt, ein sinnvolles Arbeiten mit Turbo auf dem CPC 6128 dadurch jedoch wesentlich erleichtert wird. Obwohl das eigentlich im Interesse des Turbo-Programmierers liegt, finden Sie die folgenden Erklärungen in keinem Handbuch.

13.1 Die ESC-Funktionen von Turbo

Turbo bedient sich – ähnlich wie bei der Ansteuerung eines Druckers – sogenannter ESC-Funktionen, die ihren Namen daher beziehen, weil ein **Write**-Befehl mit dem Zeichen Nummer 27, dem Code für ESC, eingeleitet wird. Folgen dahinter bestimmte weitere Zeichen, dann wird kein tatsächlicher **Write**-Befehl ausgeführt, sondern es erfolgt eine Umschaltung.

Welche Möglichkeiten dadurch eröffnet werden, die Sie mit den gewöhnlichen Turbo-Befehlen niemals haben, sehen Sie in der Aufstellung im folgenden Abschnitt.

13.2 Anwendung der ESC-Funktionen – Auflistung

Grundsätzlich erfolgt eine Anwendung dieser Funktionen durch einen **Write**-Befehl, der mit **Write**(#27,...) beginnt, wobei die Zeichen #27 für CHR(27) stehen. Danach folgt ein weiteres Zeichen, und eventuell ein oder mehrere Parameter, die die Art und Weise der Funktion auswählen. Dieses Zeichen kann als Codenummer (z.B. #48) oder mit der Turbo-Anweisung **Chr**(48) oder dem entsprechenden ASCII-Zeichen ausgegeben werden.

Nicht alle Folgezeichen nach einem #27 bewirken eine Sonderfunktion. Wir listen Ihnen hier die auf, die wir für brauchbar und nützlich halten:

Ausschalten der Drive-Anzeige (rechts unten):

```
Write(#27,#48); oder Write(#27,'0');
```

Einschalten der Drive-Anzeige:

```
Write(#27,#49); oder Write(#27,'1');
```

Auswahl des Zeichensatzes:

```
Write(#27,#50,XX)
```

mit XX als Parameter für einen aus acht Zeichensätzen z.B.

XX='0' Standard (USA)
XX='1' Frankreich
XX='2' Deutschland
XX='3' England

Sie brauchen also nicht auf die Umlaute oder ein »ß« zu verzichten, wenn Sie in Ihrem Programm **Write**(#27,'2','2'); aufrufen.

Auswahl des Bildschirmmodus:

`Write(#27,#51,'0');`	für 20-Zeichen-Modus	Mode(0)
`Write(#27,#51,'1');`	für 40-Zeichen-Modus	Mode(1)
`Write(#27,#51,'2');`	für 80-Zeichen-Modus	Mode(2)

Cursor eine Zeile nach oben bewegen (Cursor up):

```
Write(#27,#65); oder Write(#27,'A');
```

Cursor eine Zeile nach unten bewegen (Cursor down):

```
Write(#27,#66); oder Write(#27,'B');
```

Cursor eine Spalte nach links bewegen (Cursor left):

```
Write(#27,#67); oder Write(#27,'C');
```

Cursor eine Spalte nach rechts bewegen (Cursor right):

```
Write(#27,#68); oder Write(#27,'D');
```

Bildschirm löschen ohne Cursorbewegung:

```
Write(#27,#69); oder Write(#27,'E');
```

Cursor Home:

```
Write(#27,#72); oder Write(#27,'H');
```

Alle Zeichen des Schirms hinter dem Cursor löschen:

```
Write(#27,#74); oder Write(#27,'J');
```

Alle Zeichen der Zeile hinter dem Cursor löschen:

```
Write(#27,#75); oder Write(#27,'K');
```

Eine Zeile an der Cursorposition einfügen:

```
Write(#27,#76), oder Write(#27,'L');
```

Delline, eine Zeile löschen:

```
Write(#27,#77); oder Write(#27,'M');
```

Del, ein Zeichen löschen:

```
Write(#27,#78); oder Write(#27,'N');
```

Grafikcursor pixelweise plazieren:

```
Write(#27,#89,#x,#y); oder Write(#27,'Y',Chr(x),Chr(y));
```

Dabei sind beide Koordinaten jeweils um 31 zu erhöhen, um die absolute Bildschirmposition zu erreichen. Siehe dazu auch Kapitel 17 (**Inline**-Beispiel).

Alle Zeichen vor dem Cursor löschen:

```
Write(#27,#100); oder Write(#27,'d');
```

Cursor einschalten (sichtbar):

```
Write(#27,#101); oder Write(#27,'e'),
```

Cursor ausschalten (unsichtbar):

```
Write(#27,#102); oder Write(#27,'f');
```

Zeilenanfang löschen:

```
Write(#27,#111); oder Write(#27,'o');
```

Schrift auf revers schalten (Revers on):

```
Write(#27,#112); oder Write(#27,'p');
```

Schrift auf normal schalten (Revers off):

```
Write(#27,#113); oder Write(#27,'q');
```

Grafikschirm ausschalten:

```
Write(#27,#120); oder Write(#27,'x');
```

Grafikschirm einschalten:

```
Write(#27,#121); oder Write(#27,'y');
```

Hintergrundfarbe wählen:

```
Write(#27,#99,#x); oder Write(#27,'c',Chr(x));
```

Zeichenfarbe wählen:

```
Write(#27,#98,#x); oder Write(#27,'b',Chr(x));
```

Einige dieser Funktionen wenden wir nun an, um unser Würfelspiel mit einem Titelbild beginnen zu lassen, das etwas Aufmerksamkeit erregt.

13.3 Umschalten des Bildschirmmodus Programm P35.DEM: Modusdemo

Beginnen wir damit, daß wir erst einmal lernen, von Turbo aus den Bildschirmmodus zu wechseln. Dazu lassen wir auf dem Schirm einen Schriftzug erscheinen, bei dem sich in rhythmischen Abständen die Größe der Schriftzeichen ändert.

Für unser Demonstrationsprogramm benötigen wir als erstes eine Prozedur, die uns einen frei wählbaren Modus einschaltet. Wir geben ihr den Parameter »modus« mit, der in einem String den gewünschten Umschaltwert übergibt.

Als nächstes schreiben wir ein Unterprogramm, das uns die verschiedenen Modi vorführt. Im Hauptprogramm lassen wir uns noch nach einem gewünschten Bildschirmmodus abfragen.

```
      PROGRAM modus_demo; (* P35.DEM *)
      TYPE
(*A*)   param=String[1];
      VAR
        modus,schrift:param;
(*B*) PROCEDURE moduswechsel (modus:param);
      BEGIN
        Write(#27,'3',modus);
        Write(#27,'0');
      END;
(*C*) PROCEDURE modusdemo;
      BEGIN
        Write(#27,'f');
        REPEAT
          moduswechsel('2');GotoXY(32,15);
          Write('DREHWUERFEL');Delay(300);
          moduswechsel('1');GotoXY(12,14);
          Write('DREHWUERFEL'); Delay(300);
```

```
          moduswechsel ('0');GotoXY(5,13);
          Write('DREHWUERFEL');Delay(300);
        UNTIL KeyPressed;
        Write(#27,'e');
      END;
      BEGIN
        ClrScr;
        modusdemo;
(*D*)   moduswechsel('1');
        WriteLn(^J^J^M,'Aus drei Modi koennen Siewaehlen:'^J^J^M);
        WriteLn(^J^M'0 = 20 Zeichen');
        WriteLn(^J^M'1 = 40 Zeichen');
        WriteLn(^J^M'2 = 80 Zeichen'^J^J^M);
        Write('Welcher Modus wird gewuenscht?   ');
        ReadLn(modus);
        moduswechsel(modus);
        GotoXY(5,10);Write('DREHWUERFEL');
        REPEAT UNTIL KeyPressed;
(*E*)   moduswechsel('2');
      END.
```

Erläuterungen:

(A) Die Variable »modus« ist vom Typ **String**[1]. Man könnte hier auch als Typ **Char** wählen, da immer nur ein Zeichen übergeben wird.

(B) Die Prozedur »moduswechsel« übernimmt den Parameter und setzt den gewünschten Modus, schaltet aber auch gleichzeitig die Drive-Anzeige aus.

(C) In der Prozedur »modusdemo« schalten wir zunächst den Cursor aus und lassen in einer **REPEAT**-Schleife alle drei Modi so lange laufen, bis eine Taste gedrückt wird. Zu beachten ist, daß die **GotoXY**-Anweisung sich bei jedem Moduswechsel automatisch auf die Zeichengröße einstellt, so daß wir z.B. in Modus(0) nur 20 Spalten anwählen können.

(D) Wenn wir genug vom Wechsel gesehen haben, schalten wir auf den 40-Zeichen-Schirm um und lassen in einem kleinen Menü den Modus neu per Hand einstellen.

(E) Vorsichtshalber schalten wir vor dem Ende noch schnell auf den 80-Zeichen-Schirm zurück.

13.4 Auswählen des Zeichensatzes Programm P36.DEM: Schriftdemonstration, Umlaute

Um Ihnen zu zeigen, wie Turbo die Zeichensätze für die einzelnen Länder verwaltet, führen wir wieder mit einem kleinen Demo-Programm vor, was alles möglich ist. In einer Umschaltprozedur geben wir über einen Parameter an, welcher Zeichensatz gewünscht wird. Und in einem Demo-Teil führen wir die kritischen Zeichen vor.

Probieren Sie das mal aus. Sie werden feststellen, daß man auch in einer einzigen Zeile die Zeichen mehrerer Sätze verwenden kann.

```
      PROGRAM Schrift_demo; (* P36.DEM *)
      TYPE
        param=String[2];
      VAR
        schrift:param;
(*A*) PROCEDURE schriftwechsel(schrift:param);
      BEGIN
        Write(#27,'2',schrift);
      END;
(*B*) PROCEDURE schriftdemo;
      VAR i:Integer;
      BEGIN
        FOR i:=48 TO 70 DO
        BEGIN
          schrift:=Chr(i);schriftwechsel(schrift);
          WriteLn(i-48,'.  @[]\'}{#| \fter W}rfel {ndern!');
        END;
      END;
      BEGIN
        ClrScr;
        schriftdemo;
(*C*)   REPEAT UNTIL KeyPressed;
        schriftwechsel('2');
(*D*)   WriteLn(^J^J^M,'Aus 4 Saetzen koennen Sie waehlen:'^J^J^M);
        WriteLn(^J^M'0 = Standard');
        WriteLn(^J^M'1 = Franzoesisch');
        WriteLn(^J^M'2 = Deutsch');
        WriteLn(^J^M'3 = Englisch'^J^J^M);
        Write('Welcher Modus wird gewuenscht?   ');
        ReadLn(schrift);
        schriftwechsel(schrift);
(*E*)   GotoXY(1,10); Write('{{|||\\\@@@ DREHWUERFEL @@@[[[]]]}}');
        REPEAT UNTIL KeyPressed;
        schriftwechsel('0');
      END.
```

Erläuterungen:

(A) Mit dem Parameter »schrift« schalten wir auf den neuen Zeichensatz um.

(B) Die Schriftdemonstration lassen wir von 48 bis 70 laufen, was den Zeichen '0' bis 'F' entspricht. Sie sehen dann, wie die kritischen Zeichen in der jeweiligen Sprache ausgegeben werden.

(C) Nach Abbruch der Demonstration mit einer beliebigen Taste schalten wir auf den deutschen Satz um.

(D) Nach freier Wahl können Sie nun einen bestimmten Zeichensatz auswählen. Von den acht möglichen haben wir vier im Menü vorgestellt, was Sie aber nicht daran hindern sollte, auch einmal die Zahl 17 einzutippen, denn eine Absicherung dagegen ist nicht eingebaut.

(E) Mit einer kleinen Demo-Zeile beenden wir das Programm, schalten aber vorher schnell noch auf den Standard-Zeichensatz um, mit dem Sie normalerweise programmieren sollten, weil sonst die eckigen und geschweiften Klammern nicht greifbar sind.

13.5 Umschalten der Bildschirmfarben Programm P37.DEM: Hintergrund-, Zeichenfarbe, geschachtelte Prozeduren

Da auch der Monochromschirm des CPC 6128 »Farben« in Form von Grüntönen zuläßt, lassen wir Turbo damit ein wenig herumspielen, um zu demonstrieren, daß sich Hintergrund- und Zeichenfarbe beliebig variieren lassen.

Die Prozedur »vielfarben« wechselt im Rhythmus sowohl den Code für den Background als auch für die dargestellten Zeichen. Dazu bedient sie sich der beiden Prozeduren »farbwechselH« und »farbwechselZ«. Diese Unterprogramme sind innerhalb der Prozedur »vielfarben« deklariert. Wir sprechen deshalb von verschachtelten Prozeduren.

Für die Farbwerte gilt zu beachten, daß sie beim Durchlaufen des Parameters nicht von 0 bis 255 gleichmäßig heller oder dunkler werden, sondern Sprünge machen.

Ein paar Beispiele aus der Helligkeitsbestimmung:

Für die Farbwerte gilt z.B.

0 = mittelhell
32 = dunkel
21 = hell

Über 32 folgt dann wieder der Wechsel auf mittelhell usw. Deswegen betrachten wir bei unserem Demo-Programm nur einen Ausschnitt aus der Farbskala.

```
      PROGRAM Farbendemo; (* P37.dem *)
      TYPE
(*A*)   param=Char;
      VAR
        i:Integer;

(*B*) PROCEDURE Moduswechsel (modus:param);
      BEGIN
        Write(#27,'3',modus,#27,'0');
      END;

      PROCEDURE Schriftwechsel (schrift:param);
      BEGIN
        Write(#27,'2',schrift);
      END;

      PROCEDURE vielfarben; (* VIELFARB.I37 *)
(*C*) VAR
      hinterg, zeichenf,i:Integer;
      PROCEDURE farbwechselH(hifarbe:Integer);
      BEGIN
        Write(#27,'c',Chr(hifarbe));
      END;
      PROCEDURE farbwechselZ(zeifarbe:Integer);
      BEGIN
      Write(#27,'b',Chr(zeifarbe));Delay(30);
      END;
(*D*) BEGIN
        hinterg:=32;zeichenf:=32;
        REPEAT
(*E*)     hinterg:=hinterg+1;
          zeichenf:=zeichenf-1;
          IF hinterg>47 THEN hinterg:=12;
          IF zeichenf<17 THEN zeichenf:=52;
          farbwechselH(hinterg);
```

```
          farbwechselZ(zeichenf);
        UNTIL KeyPressed;
(*F*)   Write(#27,'c',Chr(32),#27,'b',Chr(22));
      END;
      BEGIN
(*G*)   moduswechsel('0');schriftwechsel('2');
        ClrScr;GotoXY(1,6);
        FOR i:=1 TO 10 DO
        WriteLn('####  WUERFELN  #####');
        vielfarben;
        moduswechsel('2');schriftwechsel('0');
      END.
```

Erläuterungen:

(A) Der Typ »param« wird für die Include-Prozeduren benötigt.

(B) Wir bedienen uns der oben besprochenen Umschaltungen für Bildschirmmodus und Zeichensatz. Aus den Demoprogrammen haben wir die Umschaltteile »PROCEDURE Moduswechsel« und »PROCEDURE Schriftwechsel« herauskopiert und etwas kürzer gefaßt.

(C) »hinterg« und »zeichenf« sind lokale Variablen, die aber für die folgenden eingeschachtelten Prozeduren »farbwechselH« und »farbwechselZ« auch Gültigkeit hätten. Dort werden aber die Daten mit Festwertparametern übernommen.

(D) Der Durchlauf beginnt mit »Verdunklung«, weil Hintergrund und Zeichen auf Schwarz gestellt werden.

(E) Sodann erhöhen wir den Code für den Hintergrund und erniedrigen gleichzeitig den Code für die Zeichenfarbe, begrenzen aber die Bereiche von 12 bis 48 bzw. von 16 bis 52. Diese Grenzwerte sollten Sie einmal verändern.

(F) Beim Abbruch durch einen Tastendruck schalten wir vorsichtshalber den Bildschirm wieder so ein, daß man Hintergrund und Zeichen voneinander unterscheiden kann.

(G) Der eigentliche Start beginnt mit Umschalten auf den 20-Zeichen-Schirm und den deutschen Zeichensatz. Es folgt eine Schrift und beim Beenden des Programms die Herstellung der normalen Einstellungen.

Achtung: Wenn Sie mit diesem Programm experimentieren, kann es sein, daß sich plötzlich nichts mehr auf dem Schirm rührt. Der Rechner ist dann nicht etwa abgestürzt, sondern Sie sind vermutlich an eine Einstellung der Farben geraten, bei der man Hintergrund und Zeichen nicht mehr auseinanderhalten kann. Sie sollten dann ein Miniprogramm auf Diskette bereithalten, das wieder in die Standardeinstellung zurückschaltet, sonst müssen Sie Turbo immer wieder neu starten.

13.6 Ein Titelblatt für das Würfelspiel PROCEDURE P38: Farbwechsel, Moduswechsel

Als endgültigen Abschluß für unser Spiel mit dem Drehwürfel setzen wir ein Titelblatt in Szene, das die eben behandelten Turbo-Möglichkeiten ausnützt. Wir legen diesen Programmteil gleich als Prozedur aus und können ihn auf diese Weise dann leicht in das Hauptprogramm einbinden.

Um den Quelltext nicht zu sehr aufzublähen, beschränken wir uns auf ein relativ einfaches Bildschirmmuster, in das wir die notwendigsten Informationen für das Spiel schreiben. Der Fantasie sind hier keine Grenzen gesetzt.

Schauen Sie sich vielleicht erst einmal den Turbo-Text an, dann werden Sie sicher neue (natürlich auch bessere) Ideen haben:

```
      PROCEDURE titelbild; (* P38.INC *)
      TYPE
(*A*)   feld=Array [1..25] Of String[40];
        zeiger=feld;
        l40=String[40];
        l22=String[22];
        l9=String[9];
        param=Char;
      CONST
        leer:l22='                      ';
        muster:l22='**********************';
        rand:l9='*********';
      VAR
        zeile:zeiger;
        lang:l40;
        i:Integer;
(*B*) PROCEDURE schirmtext;
        BEGIN
          lang:=muster+rand+rand;
```

```
          FOR i:=1 TO 22 DO zeile[i]:=lang;
          FOR i:=5 TO 8 DO zeile[i]:=rand+leer+rand;
          zeile[9]:=rand+'   DREH den Wuerfel   '+rand;
          FOR i:=10 TO 16 DO zeile[i]:=rand+leer+rand;
          zeile[12]:=rand+'   mit den Tasten:    '+rand;
          zeile[14]:=rand+' <R> <O> <V>  <Z> <E> '+rand;
          zeile[15]:=rand+' <L> <R> <H>  <W> <Q> '+rand;
        END;
      BEGIN
(*C*)   Write(#27,'3','1',#27,'H',#27,'0');
        schirmtext;
(*D*)   Write(#27,'b',Chr(0),#27,'c',Chr(0));
        FOR i:=1 TO 22 DO Write(zeile[i]);
        GotoXY(3,25);Write('Zum Start beliebige Taste druecken');
        vielfarben;
(*E*)   moduswechsel('0');
      END;
```

Erläuterungen:

(A) Wir arbeiten mit dem 40-Zeichen-Schirm und legen dafür ein **Array** an. Mit verschiedenen Strings füllen wir später den Schirm.

(B) Die Prozedur »schirmtext« bereitet die Strings für das Beschreiben des Schirms auf.

(C) Wir können auch mehrere ESC-Funktionen zugleich in einer einzigen **Write**-Anweisung unterbringen. In unserem Fall bedeutet das:

- Schalte auf Modus 1
- Cursor Home
- Drive-Anzeige aus

(D) Wenn Sie wollen, können Sie den Bildschirm auch in abgedunkeltem oder hellem Modus beschreiben lassen und erst auf »Kontrast« schalten, wenn die Schriftzüge stehen.

(E) Nach Abschluß der Farbwechselroutine wird auf den 40-Zeichen-Schirm geschaltet, in dem der Würfel dann gespielt werden kann.

Aufgaben:

1. Hätten Sie nicht Lust, für unser Spiel zwei Titelseiten zu entwerfen, die im Wechsel den Würfel darstellen und eine kurze Spielanleitung bieten?

2. Auch ein Würfel, der seine Konfiguration ändert, wäre als erste Seite auf dem Schirm recht nett.

Unser Programm für den Zauberwürfel ist nun soweit fertig, daß wir es einem Spieler anbieten können. Das folgende Listing zeigt Ihnen – nur grob kommentiert – wie der komplette Quelltext aussehen könnte. Dabei haben wir alle Prozeduren, die man normalerweise als Include-Dateien auf Diskette ablegt, noch einmal ausgedruckt. Ein paar kleine Änderungen gegenüber den Einzelteilen wurden allerdings notwendig, weil z.B. der Bildschirm während des Spiels im 20-Zeichen-Modus steht, was bei der Prozedur »ausdruck.20z« gegenüber der ursprünglichen Variante »P23.PAS« berücksichtigt wurde.

13.7 Listing Programm P39: Zauberwürfel (Spiel)

```
PROGRAM Drehwuerfel; (* Komplettlisting DREHWURF.P39 *)
TYPE
  st=String[1];
  mini=st;
  param=Char;
  reihe=Array [1..2] Of mini;
  schicht=Array [1..2] Of reihe;
  wuerfel3=Array [1..2] Of schicht;
  zeigertyp=^Integer;
VAR
  wuerfel,iwurf:wuerfel3;
  si,ri,mi,ze,sp,wh,con:Integer;
  nummer,retten:st;
  auswahl:Set Of Char;
  taste:Char;
  z1,z2,z3:zeigertyp;
  zahl:Real;
CONST
{In dieser Einstellung ist der Wuerfel in der gesuchten Lage}
fixpos:wuerfel3=((('1','2'),('3','4')),(('5','6'),('7','8')));
PROCEDURE (* ausdruck.20z - Teil 1 *) ausdruck;
```

```
VAR
  druckzeile:Integer;
BEGIN
  ze:=-8;sp:=12;
  FOR si:=1 TO 2 DO
  BEGIN
    ze:=ze+15;sp:=sp-6;
    FOR ri:=1 TO 2 DO
    BEGIN
      sp:=sp+2*ri;ze:=ze-2*ri+1;
      GotoXY(sp,ze);
      FOR mi:=1 TO 2 DO
      Write(' Nr',#27,'p',wuerfel[si,ri,mi],#27,'q');
    END;
  END;
  (* Teil 2 *)
  WriteLn(^J^J);druckzeile:=12;
  Write(#27,'q');
  FOR si:=1 TO 2 DO
  FOR ri:=1 TO 2 DO
  FOR mi:=1 TO 2 DO
  BEGIN
    GotoXY(1,druckzeile);
    Write('Nr.',wuerfel[si,ri,mi]);
    IF wuerfel[si,ri,mi]=fixpos[si,ri,mi]
    THEN WriteLn('+')
    ELSE WriteLn('-');
    druckzeile:=druckzeile+1;
  END;
  Write(^J,#27,'p','Taste:',#27,'q',' ');
END;
PROCEDURE (* P29 *) rechts_links(wh,con:Integer);
(* Drehung von Schicht rechts oder links *)
VAR z: Integer;
BEGIN
  FOR z:=1 TO wh DO
  BEGIN
    retten:=wuerfel[1,1,con];
    wuerfel[1,1,con]:=wuerfel[2,1,con];
    wuerfel[2,1,con]:=wuerfel[2,2,con];
    wuerfel[2,2,con]:=wuerfel[1,2,con];
    wuerfel[1,2,con]:=retten;
  END;
  ausdruck;
END;
PROCEDURE (* P30 *) oben_unten(wh,con:Integer);
(* Drehung der oberen oder unteren Schicht *)
VAR z:Integer;
```

```
BEGIN
  FOR z:=1 TO wh DO
  BEGIN
    retten:=wuerfel[con,1,1];
    wuerfel[con,1,1]:=wuerfel[con,1,2];
    wuerfel[con,1,2]:=wuerfel[con,2,2];
    wuerfel[con,2,2]:=wuerfel[con,2,1];
    wuerfel[con,2,1]:=retten;
  END;
  ausdruck;
END;
PROCEDURE (* P31 *) vorn_hinten(wh,con:Integer);
(* Drehung der oberen oder unteren Schicht *)
VAR z:Integer;
BEGIN
  FOR z:=1 TO wh DO
  BEGIN
    retten:=wuerfel[1,con,1];
    wuerfel[1,con,1]:=wuerfel[2,con,1];
    wuerfel[2,con,1]:=wuerfel[2,con,2];
    wuerfel[2,con,2]:=wuerfel[1,con,2];
    wuerfel[1,con,2]:=retten;
  END;
  ausdruck;
END;
PROCEDURE (* P32 *) zufall;
VAR
  zu,x:Integer;
  ziff:st;
  kette:String[8];
BEGIN
  x:=0;
  kette:='12345678';
  Randomize;
  FOR si:=1 TO 2 DO
  FOR ri:=1 TO 2 DO
  FOR mi:=1 TO 2 DO
  BEGIN
    zu:=Random(8-x)+1;
    wuerfel[si,ri,mi]:=Copy(kette,zu,1);
    iwurf[si,ri,mi]:=wuerfel[si,ri,mi];
    Delete(kette,zu,1);
    x:=x+1;
  END;
  ausdruck;
END;
PROCEDURE moduswechsel (* ModusW.I35 *) (modus:param);
BEGIN
```

```
  Write(#27,'3',modus);
  Write(#27,'0');
END;
PROCEDURE vielfarben; (* vielfarb.I37 *)
VAR
  hinterg,zeichenf,i:Integer;
PROCEDURE farbwechselH(hifarbe:Integer);
BEGIN
  Write(#27,'c',Chr(hifarbe));
END;
PROCEDURE farbwechselZ(zeifarbe:Integer);
BEGIN
  Write(#27,'b',Chr(zeifarbe));Delay(30);
END;
BEGIN
  hinterg:=32;zeichenf:=32;
  REPEAT
    hinterg:=hinterg+1;
    zeichenf:=zeichenf-1;
    IF hinterg>47 THEN hinterg:=12;
    IF zeichenf<17 THEN zeichenf:=52;
    farbwechselH(hinterg);
    farbwechselZ(zeichenf);
  UNTIL KeyPressed;
  Write(#27,'c',Chr(96),#27,'b',Chr(22));
END;
PROCEDURE titelbild; (* P38 *)
TYPE
  feld=Array [1..25] Of String[40];
  zeiger=feld;
  L40=String[40];
  L22=String[22];
  L9=String[9];
  param=Char;
CONST
  leer:l22='                      ';
  muster:l22='**********************';
  rand:l9='*********';
VAR
  zeile:zeiger;
  lang:l40;
  i:Integer;
PROCEDURE schirmtext;
BEGIN
  lang:=muster+rand+rand;
  FOR i:=1 TO 22 DO zeile[i]:=lang;
  FOR i:=5 TO 8 DO zeile[i]:=rand+leer+rand;
  zeile[9]:=rand+'   DREH den Wuerfel   '+rand;
```

```
  FOR i:=10 TO 16 DO zeile[i]:=rand+leer+rand;
  zeile[12]:=rand+'   mit den Tasten:    '+rand;
  zeile[14]:=rand+' <R> <O> <V>  <Z> <E> '+rand;
  zeile[15]:=rand+' <L> <R> <H>  <W> <Q> '+rand;
END;
BEGIN
  Write(#27,'3','1',#27,'H',#27,'0');
  schirmtext;
  Write(#27,'b',Chr(0),#27,'c',Chr(0));
  FOR i:=1 TO 22 DO Write(zeile[i]);
  GotoXY(3,25);Write('Zum Start beliebige Taste druecken');
  vielfarben;
  moduswechsel('0');
END;
PROCEDURE (* P34 *) wartezahl;
BEGIN
  REPEAT
  z1^:=z1^+1;
  IF z1^=10000 THEN
  BEGIN
    z1^:=0;
    z2^:=z2^+1;
    IF z2^>10000 THEN
    BEGIN
      z2^:=0;
      z3^:=z3^+1;
    END;
  END;
  UNTIL KeyPressed;
END;
{Wenn die gleiche Zufallsstellung noch mal aufgerufen werden
 soll, wird die gerettete Stellung 'iwurf' wieder zugewiesen}
PROCEDURE gleichwurf;
BEGIN
  FOR si:=1 TO 2 DO
  FOR ri:=1 TO 2 DO
  FOR mi:=1 TO 2 DO
  BEGIN
    wuerfel[si,ri,mi]:=iwurf[si,ri,mi];
  END;
END;
{Den Miniwuerfeln werden Nummern zugewiesen.}
PROCEDURE zuweisung;
BEGIN;
  FOR si:=1 TO 2 DO
  FOR ri:=1 TO 2 DO
  FOR mi:=1 TO 2 DO
  BEGIN
```

```
      Str((si-1)*4+(ri-1)*2+(mi-1)+1:1,nummer);
      wuerfel[si,ri,mi]:=nummer;
    END;
    ausdruck;
  END; (* zuweisung *)
  PROCEDURE menu;
  Label start1,start2;
  BEGIN
    start1:
    zuweisung;
    start2:
    ausdruck;
    (* Beginn des Menues *)
    auswahl:= ['L','R','O','U','V','H','E','A','Z','Q','W'];
    taste:='X';
    WHILE taste <> 'E' DO
    BEGIN (* Beginn der Menueschleife *)
      wartezahl;
      Read (KBD, taste);
      IF UpCase(taste) IN auswahl THEN
      BEGIN
        Write(UpCase(taste));
        IF Ord(taste) < 96 THEN wh:=3
        ELSE wh:=1;
        CASE UpCase(taste) OF
          'O': BEGIN con:=1;oben_unten(wh,con);END;
          'U': oben_unten(wh,2);
          'L': rechts_links(wh,1);
          'R': rechts_links(wh,2);
          'V': vorn_hinten(wh,1);
          'H': vorn_hinten(wh,2);
          'Q': BEGIN moduswechsel('2');Exit;END;
          'A': Goto start1;
          'W': BEGIN
                 gleichwurf;z1^:=0;z2^:=0;z3^:=0;
                 Goto start2
               END;
          'E': BEGIN
                 zahl:=(z3^*1.0E8+z2^*1.0E4+z1^);
                 GotoXY(1,1);
                 Write(zahl/5300:6:1,'Sek.');
                 GotoXY(8,21);
               END;
          'Z': BEGIN zufall;z1^:=0;z2^:=0;z3^:=0; END;
        END; (* Ende von CASE OF *)
      END; (* Ende der Auswahl *)
    END; (* Ende des Menues *)
  END; (* Ende der Prozedur Menue *)
```

```
BEGIN (* Hauptprogramm *)
  titelbild;
  GetMem(z1,16);GetMem(z2,16);GetMem(z3,16);
  GotoXY(1,24);
  Write('<R> <O> <V> <Z> <E>');
  GotoXY(1,25);
  Write('<L> <U> <H> <W> <Q>');
  menu;
END. (* Ende des Programms Drehwuerfel *)
```

Kapitel

14

Erstellung von Bildschirmmasken

14.1 Wichtiges über Masken und Fenster

Unter Maskierung versteht man die Abdeckung des Bildschirms bis auf ganz bestimmte Ausschnitte, sogenannte Fenster. Der Begriff Abdeckung ist dabei aber nicht wörtlich zu nehmen, sondern besagt lediglich, daß der abgedeckte Bereich für den Benutzer nicht zugänglich ist.

Die Nützlichkeit der übrigen, einmal festgelegten offenen Stellen ist offensichtlich: Für eine bestimmte Eingabe, z.B. eines Ortsnamens, wird ein Zeilenstück auf dem Bildschirm bestimmt, in welchem das Eingabe-Echo der Tastatur erscheint.

Der Anwender kann nun diese Eingabe in der gewünschten Form vornehmen und innerhalb des bereitgestellten Ausschnitts beliebige Positionen anspringen, er kommt aus diesem Feld nicht heraus (falls der Programmierer nichts übersehen hat!).

Bestimmt erinnern Sie sich an unsere anfänglichen Versuche mit der **Read**-Anweisung, bei der man mit **BufLen** ebenfalls die Länge der Eingabe begrenzen konnte. Das ist

zwar richtig, hat aber mit einer Maskierung noch nicht viel gemeinsam. Denn beim Einlesen mit **Read** können Sie den Cursor nicht überall dorthin steuern, wo Sie wollen, ohne den bereits vorhandenen Text zu zerstören. Und vor allem nimmt **Read** jedes Zeichen sofort ungeprüft von der Tastatur auf und stellt es auf dem Bildschirm dar.

Mit Hilfe von Maskenausschnitten haben wir aber eine ganze Reihe von Möglichkeiten, erstens komfortable und zweitens richtige Eingaben zu erreichen:

- Jeder Ausschnitt der Bildschirmmaske wird so angelegt, daß er genau auf die Länge der gewünschten Eingabe zugeschnitten ist.

- Für jeden Ausschnitt der Maske werden nur die Zeichen zur Eingabe zugelassen, die erwünscht sind.

- Innerhalb des Ausschnitts kann der Anwender den Cursor an beliebige Positionen steuern.

- Es muß möglich sein, fehlerhafte Zeichen zu korrigieren. Löschen und Überschreiben, Einfügen und Zusammenziehen des Inhalts müssen ermöglicht werden.

- Jede Eingabe wird mit einer bestimmten Taste beendet.

- Wenn es notwendig ist, wird die Eingabe auf die richtige Form überprüft. Wird sie nicht vorgefunden, dann wird der Weitersprung zum nächsten Ausschnitt nicht zugelassen. Das ist besonders dann wichtig, wenn die eingetippte Zeile Daten enthält, die unter einer besonderen Form weiterverarbeitet werden. Es ist nämlich einfacher, den Anwender zur richtigen Eingabe zu bringen, als hinterher per Programm auf alle möglichen und unmöglichen Eingabeformen zu reagieren. Denken Sie nur daran, auf wieviele Arten allein ein Datum geschrieben werden kann:

  1.2.1945 oder 1.2.45 oder 01.02.45 oder 01/02/45 oder 010245 ...

- Und wenn man professionell arbeiten will – und hier wollen wir wirklich in die Nähe der Profis kommen – muß man dem Anwender Gelegenheit geben, jeden Maskenausschnitt beliebig oft anspringen zu können.

- Mittels einer bestimmten Abschlußtaste bestätigt der Benutzer, daß er bereits alle Ausschnitte bearbeitet hat.

- Die Bedienung der Tastatur sollte sich nach bereits bestehenden Normen richten

oder wenigstens daran anlehnen, denn wenn man z.B. bereits die Turbo-Tastatur im Griff hat, ist es verwirrend, wenn sich bei Masken plötzlich der Cursor nicht mehr dorthin bewegt, wo man ihn eigentlich haben wollte.

Sie sehen, die Problematik ist recht umfangreich. Aber mit Turbos und unserer Hilfe werden Sie den richtigen »Dreh« mit den Fenstern schnell herausfinden.

Übrigens: Um für eine klare Sprachregelung zu sorgen, verstehen wir unter einer Maske den Gesamtaufbau des Bildschirms, der formatierte Eingaben erlaubt. Es gefällt uns gar nicht, wenn in anderen Veröffentlichungen unter einer Maske nur ein einzelner Eingabebereich gemeint ist. Das ist für uns ein Fenster oder ein Maskenausschnitt.

14.2 Erstellung einer Eingabemaske Programm P40.DEM: Eingabeschleife, Prozeduraufrufe

Wir setzen uns als Ziel für den Aufbau einer Bildschirmmaske: Fünf Eingaben sollen in je einem Maskenausschnitt aufgenommen werden. Da zu jeder Eingabe eine höchstzulässige Länge gehört, ordnen wir diese in einem **Array** an, das wir »maxlen« nennen wollen.

Die Anfangspositionen der einzelnen Ausschnitte halten wir in zwei Feldern »xpos« und »ypos« fest, in denen wir die Spalte bzw. die Zeile ablegen. Dafür ist die Prozedur »maskenposition« zuständig.

Jedem Maskenausschnitt wird auf dem Schirm eine Bemerkung vorangestellt, aus der ersichtlich ist, was dort eigentlich als Eingabe erwartet wird. Dies macht die Prozedur »notizen«.

Zur Eingabe selbst wird jeder Ausschnitt in der festgelegten Reihenfolge angesprungen. Als Abschlußtaste legen wir die Rücklauftaste **<RETURN>** fest. Sie erzeugt auch jeweils den Sprung zum nächsten Eingabefeld. Nach der fünften und letzten Eingabe erfolgt der Sprung zurück zur ersten, so daß alle Eingaben beliebig oft wiederholt oder korrigiert werden können.

Die Hauptarbeit übernimmt eine Prozedur, die wir »Bearbeitung« genannt haben. Sie überprüft und analysiert jedes einzelne Zeichen der Eingaben und reagiert entsprechend unseren Vorstellungen.

Wird **<CLR>** oder **<ESC>** gedrückt, so bedeutet das, daß alle Eingaben beendet sind. Der Inhalt der Maskenausschnitte steht danach in einem **Array** zur weiteren Bearbeitung bereit. Damit wir unser Ergebnis auch bewundern lassen können, fügen wir noch einen Testausdruck an.

Unser Hauptprogramm können wir somit schon einmal zusammenstellen:

```
      PROGRAM Eingabe_Maske; (* P40.DEM *)
      TYPE
(*A*)   m=Array [1..5] Of Integer;
        l80=String[80];
      VAR
(*B*)   i,cursor,tl:Integer;
          ch:Char;
          eingabe:Array [1..5] Of l80;
          xpos,ypos,maxlen:m;
        {$I maskposi.i41}
        {$I notizen.i42}
        {$I input.i43}
        BEGIN
(*C*)     ClrScr;Write(#27,'2','2');
          FOR i:=1 TO 5 DO eingabe[i]:='';
          maskenposition;
          notizen;
          i:=0;
(*D*)   {Anspringen der Ausschnitte}
        REPEAT
          i:=i+1;
          IF i>5 THEN i:=1;
          cursor:=0;
          tl:=0;
          GotoXY(xpos[i],ypos[i]);
(*E*)     {Beschreiben der Ausschnitte}
          REPEAT
            Read(KBD,ch);
            bearbeitung(ch,maxlen[i],xpos[i],ypos[i],cursor);
          UNTIL Ord(ch) IN [13,16,27,252];
        UNTIL (ch=#16) OR (ch=#252);
(*F*)   {Testausdruck}
        GotoXY(1,24);DelLine;
        GotoXY(1,18);
        FOR i:=1 TO 5 DO
        WriteLn(eingabe[i]);
        Write(#27,'2','0');
      END. {Ende des Hauptprogramms}
```

Erläuterungen:

(A) »m« ist ein **Integer-Array** mit jeweils 5 Komponenten. In den Komponenten von Feldern dieses Typs werden die Bildschirmpositionen gespeichert.

(B) Die Integervariablen »i«, »cursor« und »tl« sind Zähler, die vor allem zur Überwachung der Cursorposition und der tatsächlichen Länge des Eingabestrings (tl) verwendet werden:

»cursor« merkt sich die Stelle im Maskenausschnitt, auf der der blinkende Cursor steht. Er beginnt mit Null zu zählen und steht z.B. nach Eingabe von vier Zeichen auf dem Wert 4.

»tl« merkt sich die tatsächliche Länge der bereits eingegebenen Zeichenkette und darf nie größer werden als die maximale Länge. Die Variable »tl« ist deswegen notwendig, weil der Wert von »cursor« nicht mit der aktuellen Stringlänge übereinstimmen muß. Wir wollen doch mit dem Cursor beliebig im Maskenausschnitt hin- und herfahren können.

»xpos« und »ypos« geben in Spalte und Zeile die Anfangsposition der jeweiligen Eingabe auf dem Schirm an. Die Eingabelänge wird durch »maxlen« begrenzt.

Die notwendigen Prozeduren »maskenposition«, »notizen« und »bearbeitung« binden wir mit Include-Anweisungen ein. Sie werden extra besprochen.

(C) Dieser kleine Abschnitt dient der Initialisierung, also der Vorbereitung des Bildschirms für die gewünschten Operationen: Die Anfangspositionen der Maskenausschnitte werden aufgenommen und die Eingabebemerkungen notiert. Das Ganze erledigen wir im deutschen Zeichensatz.

(D) Beginnend mit 1 wird nun Ausschnitt für Ausschnitt angesprungen, und zwar so lange, bis entweder **<CLR>** oder **<ESC>** gedrückt wird. Das erzeugt den Code mit der Nummer 16 bzw. 252.

Damit der Cursor auch die richtige Bildschirmposition trifft, dafür sorgt der **GotoXY**-Befehl, der die zuvor festgelegten Zeilen- und Spaltenwerte des betroffenen Feldes enthalten muß.

(E) Die komplizierteste – aber durchaus überschaubare – Arbeit beginnt aber erst, nachdem nun Zeichen für Zeichen mit der **Read**-Anweisung über die Tastatur

(**KBD**) aufgenommen wird. Jedes Zeichen wird untersucht, wozu neben dem Zeichen »ch« selbst, auch die Parameter »maxlen« (für die maximale Stringlänge der aktuellen Eingabe), die Anfangspositionen »xpos« und »ypos« sowie die momentane Cursorposition »cursor« mit dem Index für den gerade bearbeiteten Ausschnitt benötigt werden.

Tritt Code 13,16 oder 252 auf, wird die Eingabe des aktuellen Feldes beendet. Bei Code 13, der mit dem einfachen **<RETURN>** erzeugt wird, erfolgt nur der Aussprung aus der **REPEAT**-Schleife (*E*), während **<CLR>** oder **<ESC>** auch aus der übergeordneten **REPEAT**-Schleife bei (*D*) führt und damit den gesamten Eingabeprozeß beendet.

(F) Nach Abschluß aller Eingaben stehen diese in einem **String-Array** »eingabe« zur weiteren Verarbeitung bereit, hier zum Testausdruck.

So viel haben Sie sicher schon gelernt, daß Sie nicht versuchen, dieses Programm laufen zu lassen. Denn etliche der Prozeduren sind ja noch gar nicht deklariert. Es sind diejenigen, die wir im Hauptprogramm als Include-Dateien ausgewiesen haben. Und das Erstellen dieser Prozeduren ist Aufgabe der nächsten Abschnitte.

14.3 Prozeduren zur Maskenbearbeitung

14.3.1 Festlegung der Positionen der Maskenausschnitte Prozedur: »masken_position« (MaskPosi.I41)

Die erste Prozedur, die das Hauptprogramm aufruft, muß die Anfangspositionen der Maskenausschnitte festlegen. Wir haben uns auf fünf Ausschnitte beschränkt und legen sie so, daß sie alle in der zwanzigsten Spalte beginnen. Zwischen jede Eingabezeile schieben wir eine Leerzeile.

Für die maximalen Eingabelängen setzen wir Werte von 25 bis 45 Zeichen fest.

```
      PROCEDURE maskenposition; (* MaskPosi.I41 *)
      BEGIN
        FOR i:=1 TO 5 DO
(*A*)   BEGIN
          xpos[i]:=20;
          ypos[i]:=2*i+5;
```

```
(*B*)       maxlen[i]:=20+5*i;
          END;
        END;
```

Erläuterungen:

(A) Die Spaltenzahl ist konstant. Die Zeilenzahl wird mit Hilfe des Schleifenzählers erhöht. Als erste Eingabezeile wird hier die siebte Bildschirmzeile angenommen.

(B) Die maximale Eingabelänge wird jeweils um 5 erhöht.

14.3.2 Ausdruck der Eingabekommentare Prozedur :"notizen« (Notizen.I42)

Welche Bemerkungen man den Eingabeausschnitten voranstellt, kommt immer darauf an, was dort eingegeben werden soll. Wir beschränken uns auf die einfache Durchnumerierung der Eingabezeilen. Außerdem geben wir am unteren Bildschirmrand eine Menüzeile aus, die die Tastenfunktionen für den Programmablauf kommentiert.

```
        PROCEDURE notizen; (* Notizen.I42 *)
        BEGIN
          FOR i:=1 TO 5 DO
          BEGIN
(*A*)       GotoXY(1,ypos[i]);
            Write('Eingabe ',i,': ');
          END;
(*B*)     GotoXY(1,24);
          Write('Schreiben: Wordstar-Tasten  <RETURN>=Sprung  ',
                '<CLR> oder <ESC>=Eingaben beenden');
        END;
```

Erläuterungen:

(A) Jeder Eingabekommentar wird vor den Maskenausschnitt auf die erste Spalte der betreffenden Zeile gesetzt.

(B) In der vorletzten Zeile sind wir großzügig und geben an, daß das Editieren der Eingaben in WordStar-Manier erfolgen kann, obwohl wir in unserem Beispielprogramm nicht alle diese Möglichkeiten vorgesehen haben. Doch Sie können ja

jederzeit weitere Funktionen einbauen, wenn Ihnen die vorgestellten nicht ausreichen.

14.3.3 Analyse und Bearbeitung der Tastatureingaben Prozedur (Input.I43) »bearbeitung«: Delete, Insert, Ctrl

Jetzt kommt die Hauptarbeit: Alle Eingaben über die Tastatur müssen überprüft werden. Wir wollen ja nicht nur Text eingeben, sondern über die Tastatur auch den Cursor in unserem Maskenausschnitt bewegen. Das erfordert etliche Festlegungen:

- Alle druckbaren ASCII-Zeichen sowie die Umlaute und das »ß« sollen als Texteingaben zugelassen werden.

- Für die Steuerung des Cursors lehnen wir uns an WordStar bzw. an Turbo an, beschränken uns aber auf die wichtigsten Funktionen. Dazu setzen wir die **<CTRL>**-Taste ein. Sie erzeugt nämlich zusammen mit einer der Eingaben »A« bis »Z« einen Codewert von 1 bis 26, so daß wir nicht extra abfragen müssen, ob die Taste **<CTRL>** und noch eine andere gedrückt wurde, sondern uns auf die Überprüfung eines einfachen Codes verlassen können. (Siehe Anhang C.)

- Im einzelnen sollen folgende Tasten Sonderfunktionen ausüben:

Taste(n)	Code-Nr.	Funktion
<CTRL/S>	19	Cursor um ein Zeichen zurück (nach links)
<CTRL/D>	4	Cursor um ein Zeichen vor (nach rechts)
<CTRL/A>	1	Cursor an den Anfang desselben Ausschnitts
<CTRL/G>	7	Zeichen an der Cursorposition wird gelöscht, alle Zeichen rechts davon rücken nach links, Cursorposition bleibt unverändert
<CTRL/H> (=BACKSPC)	8	Zeichen vor dem Cursor wird gelöscht, Cursor rückt um eins nach links, Text rutscht mit
<CTRL/T>	20	löscht die Eingabe und setzt den Cursor wieder an den Anfang des Eingabefeldes.
<CTRL/V>	22	fügt ein Leerzeichen an der momentanen Cursorposition ein, falls die maximale Eingabelänge noch nicht erreicht ist.

Wir stellen zunächst einmal die Prozedur »bearbeitung« als Ganzes vor und besprechen anschließend die einzelnen Punkte:

```
      PROCEDURE bearbeitung (* INPUT.I43 *)
(*A*)   (ch:Char;max,xp,yp:Integer;VAR cursor:Integer);
      VAR druck:Integer;
      BEGIN
(*B*)   IF ch IN [' '..'z','[','{',']','}','\','|','~'] THEN
        BEGIN  {Schreiben eines Textzeichens im Insert-Modus}
          IF (cursor<max) AND (tl<max) THEN
          BEGIN
            IF cursor>=tl THEN druck:=0 ELSE druck:=1;
            Write(ch);cursor:=cursor+1;
            Insert(ch,eingabe[i],cursor);
            tl:=tl+1;
          END;
        END
        ELSE
(*C*)   CASE Ord(ch) OF
(*D*)     19: {CTRL/S : Cursor ein Zeichen nach links}
              BEGIN
                IF cursor>0 THEN cursor:=cursor-1;
                druck:=0;
              END;
(*E*)      4: {CTRL/D: Cursor ein Zeichen nach rechts}
              BEGIN
                IF cursor<max THEN
                BEGIN
                  cursor:=cursor+1;
                  IF cursor>tl THEN
                  BEGIN
                    tl:=cursor;
                    eingabe[i]:=eingabe[i]+Chr(133);
                    druck:=1;
                  END;
                END;
              END;
(*F*)      1: {CTRL/A setzt Cursor auf Anfang des Eingabefensters}
              BEGIN cursor:=0;druck:=0;END;
(*G*)      7: {CTRL/G loescht das Zeichen unter dem Cursor}
              BEGIN
                IF tl> cursor THEN tl:=tl-1;
                Delete(eingabe[i],cursor+1,1);
                druck:=1;
              END;
(*H*)      8: {CTRL/H verkuerzt um ein Zeichen}
              BEGIN
                IF cursor>0 THEN
                BEGIN
                  Delete(eingabe[i],cursor,1);
                  cursor:=cursor-1;
```

```
                    tl:=tl-1;
                    druck:=1;
                  END;
                END;
(*I*)      22: {CTRL/V fuegt ein Zeichen ein}
                BEGIN
                  IF tl<max THEN
                  BEGIN
                    tl:=tl+1;
                    Insert(Chr(133),eingabe[i],cursor+1);
                    druck:=1;
                  END;
                END;
(*J*)      20: {CTRL/T loescht das Eingabefeld}
                BEGIN
                  cursor:=0; tl:=0;
                  eingabe[i]:='';
                  druck:=1;
                END;
          END;
(*K*)     {Kontrollausgabe der momentanen Zeilenlaenge}
          GotoXY(1,1);Write(tl:2);
          GotoXY(10,1);Write(cursor:2);
(*L*)     {Setzen des Cursors und neuer Ausdruck der Eingabe}
          IF druck=1 THEN
          BEGIN
            GotoXY(xp,yp);Write('                                    ');
            GotoXY(xp,yp);Write(eingabe[i]);
          END;
          GotoXY(xp+cursor,yp);
        END; {Ende der Procedur »bearbeitung"}
```

Erläuterungen:

(A) Die Prozedur »bearbeitung« übernimmt zunächst das eingetippte Zeichen über den Parameter »ch«, wobei eine Tastenkombination aus **<CTRL>** und einer weiteren Taste als ein einziges Zeichen interpretiert wird. Die Anfangsposition des momentan bearbeiteten Maskenausschnitts wird entsprechend den Parametern aus dem Prozeduraufruf mit »xp« und »yp« aufgefangen.

Während die bis jetzt benötigten Parameter konstante Werte enthalten, muß »cursor«, der Parameter für die Stellung des Cursors, in dem Eingabefeld variabel gehalten werden.

In der Variablen »druck« verwalten wir ein Flag, ob die Eingabe noch einmal neu gedruckt werden soll (1) oder nicht (0).

(B) Falls das angekommene Zeichen in »ch« ein einfaches Textzeichen war, also ohne **<CTRL>** eingetippt wurde, schreiben wir es in den Ausschnitt, aber nur dann, wenn die maximale Länge noch nicht erreicht ist.

Außerdem muß die Position des Cursors um eins hochgezählt werden, und die tatsächliche Länge des nun in diesem Eingabefeld enthaltenen Textstrings erhöht sich ebenfalls um eins, sofern die maximale Länge noch nicht erreicht ist.

(C) War »ch« keines dieser zugelassenen Textzeichen, dann stellen wir mit Hilfe der Standardfunktion **Ord**(ch) zuerst einmal die Code-Nummer fest, so wie sie Turbo gesehen hat.

Mit **CASE** ... **OF** sortieren wir nun die von uns selbst festgelegten Codes aus und lassen entsprechende Anweisungen los:

(D) Falls Code 19 vorgefunden wurde, müssen wir dafür sorgen, daß der Cursor von seiner gegenwärtigen Position um eine Stelle nach links rückt. Das darf er aber nur dann, wenn er schon rechts von der Anfangsposition des Eingabefeldes steht, ansonsten würde er den Maskenausschnitt verlassen, was sehr unschön wäre.

Man kann mit dem Cursor nun auch Leerzeichen überstreichen. Deswegen verringern wir auch die tatsächliche Stringlänge jeweils um eins, wenn der Cursor außerhalb der bereits gedruckten Zeichen stand.

(E) Falls Code 4 auftritt, schicken wir den Cursor um eine Position nach rechts, aber nur dann, wenn die maximale Eingabelänge noch nicht erreicht ist. Andernfalls verläßt der Blinker unkontrolliert unser Eingabefeld.

Sollte der Cursor nun rechts von bereits bestehendem Text stehen, erhöhen wir die tatsächliche Stringlänge um eins und behandeln somit Leerzeichen wie andere Textzeichen. Zum besseren Verständnis haben wir mit **Chr**(133) ein Grafikzeichen statt eines Blanks zur Ausgabe vorbereitet. Für die praktische Arbeit mit Masken wird man diese Zeile entfernen.

(F) Der einfachste Fall tritt mit Code 1 ein: Die Cursorposition wird bis zum linken Rand des Ausschnitts zurückgesetzt.

(G) Falls mit Code 7 ein Zeichen aus dem String gelöscht werden soll, verkürzen wir die tatsächliche Länge um eins und löschen mit der Standardprozedur **Delete** im entsprechenden String ein Zeichen. Der neue verkürzte String wird sofort ausgegeben. Das Druckflag »druck« läßt dies mit der Belegung 1 zu.

(H) Mit Code 8 geschieht zwar etwas Ähnliches wie gerade besprochen. Allerdings rückt hier der Cursor mit nach links, und es sieht auf dem Schirm aus, als wenn er den rechten Textteil hinter sich her zöge und damit das jeweils vor ihm liegende Zeichen überschriebe. Das darf nur so lange geschehen, bis er ganz links im Eingabefeld angelangt ist, ansonsten verbietet ihm die Anweisung **IF** cursor > 0 **THEN** ... diese Tat.

Schieben wir an dieser Stelle ein kleines Beispiel ein, damit die Sache etwas anschaulicher wird:

Nehmen wir an, im ersten Ausschnitt steht folgendes:

```
----------------------------------
Eingabe 1:         |Probetext in Zeile8...
----------------------------------
```

Die Position des Cursors sei bei der Ziffer 8, also beim 19. Zeichen dieser Zeile, gezählt vom Buchstaben »P« aus. Die tatsächliche Länge des bereits vorhandenen Strings beträgt mit den drei Punkten 22.

Nach **<CTRL/H>** sieht das Eingabefeld dann so aus:

```
----------------------------------
Eingabe 1:         |Probetext in Zeil8...
----------------------------------
```

Die tatsächliche Länge ist nun um eins kleiner, und auch die Position des Cursors hat sich nach links verrückt.

(I) Wenn Sie bis hierher alles verstanden haben, sollte Ihnen das Verfahren keine Schwierigkeiten bereiten, das angewendet werden muß, um ein Leerzeichen einzufügen, das wir in dem Demo-Programm mit dem Grafikzeichen **Chr**(133) darstellen.

Jetzt muß ab der aktuellen Cursorposition der rechte Teil um eins nach rechts verrutscht werden, aber nur, wenn die tatsächliche Länge noch kleiner ist als

die maximal zulässige. Das tun wir zunächst in dem betroffenen String, indem wir mit der Turbo-Prozedur **Insert** ein Zeichen einfügen.

Die tatsächliche Länge des Strings hat sich natürlich jetzt um eins erhöht. Der neue String muß wieder ausgegeben werden.

(J) Als letzte mögliche Eingabe wird auf Code 20 geprüft, womit die Eingabezeile wieder freigemacht werden soll.

Der entsprechende String wird geleert. Die aktuelle Stringlänge und auch die Cursorposition werden anschließend wieder auf Null gesetzt. Eine neue Eingabe wird damit vorbereitet.

(K) Zur Kontrolle der Cursorposition und der tatsächlich erreichten Stringlänge, die ja auch Leerzeichen umfassen kann, geben wir in der linken oberen Bildschirmecke den Wert von »tl« aus. Sie haben nun Gelegenheit, die Richtigkeit des Programmlaufes zu verfolgen.

Wenn Sie wollen, lassen Sie sich in WordStar- bzw. Turbo-Manier auch noch die momentane Spaltenzahl des Eingabefensters ausweisen. Sie finden sie in der Variablen »cursor«.

(L) Gleichgültig, welches Zeichen gedruckt oder welche Cursorbewegung vorgenommen wurde, mit Hilfe des **GotoXY**-Befehls wird der Cursor vor jeder weiteren Tastenaufnahme auf die Position gesetzt, die wir mit Hilfe der Variablen »cursor« stets kontrolliert haben.

Damit haben wir nun alle Cursorbewegungen abgehandelt, die wir uns vorgenommen haben. In der Regel sind sie in der Praxis völlig ausreichend, um irgendwelche Eingaben bequem unter Kontrolle zu halten. Der Komfort läßt sich natürlich weitertreiben. Aber als Programmierer sollte man auch immer den Aufwand im Verhältnis zum Nutzen sehen und nicht unbedingt jede ausgefallene Zusatzmöglichkeit zu verwirklichen versuchen. Wichtig ist nur, daß die Eingabe in der gewünschten Form zustande kommt, ohne daß der Anwender verzweifelt.

Damit sind nun auch alle Prozeduren konstruiert, die wir zum Betrieb der Maske brauchen. Wie sie ins Hauptprogramm eingefügt werden, haben wir weiter vorn eingehend besprochen. Wenn Sie das Demo-Programm ausprobieren, werden Sie vielleicht nicht ganz zufrieden sein mit dem, was Sie an Eingabekomfort vorfinden. Wir wollten aber keine Perfektion erzielen, sondern Ihnen nur ein paar Anregungen bieten. Zu verbessern gibt es immer etwas.

In abgewandelter Form eignet sich so ein Programmteil immer dann, wenn Sie dem Anwender komfortable Eingaben ermöglichen wollen. Sie können es z.B. gleich im nächsten Kapitel anwenden, wenn Sie wollen.

14.4 Manipulationen mit Stringinhalten

Weil wir gerade dabei waren, uns mit der Veränderung von Stringinhalten durch Einschieben oder Löschen einzelner Zeichen zu befassen, untersuchen wir gleich mit Hilfe von zwei schnellen Turbo-Routinen, wie man direkt auf die Speicheradressen der diversen Variablen zugreifen und somit deren Inhalte manipulieren kann.

14.4.1 Ein Versuch mit FillChar Programm P44: einfache Variablen, seltsame Ergebnisse

Turbo kennt einen Befehl, der von Beginn einer bestimmten Position an, die mit Hilfe einer »Variablen« angegeben wird, im Speicher eine bestimmte »Anzahl« Bytes einer bestimmten »Art« auffüllt.

Diese Prozedur wird so aufgerufen: **FillChar**(variable,zahl,art);

Dabei ist mit »variable« die Stelle im Speicher gemeint, wo die Variable beginnt.

Ein etwas eigenartiges Ergebnis liefert der folgende Versuch, dessen Aufbau wir gleich beschreiben:

(A) Wir legen eine Stringvariable fest, die maximal 80 Zeichen umfassen kann, bauen aber nur 20 Zeichen ein, die wir mit Ziffern füllen, damit wir später die veränderten Stellen besser ablesen können.

(B) Mit der Anweisung **FillChar**(test,10,'*') versuchen wir nun, die ersten 10 Zeichen dieses Strings durch Sternchen zu ersetzen.

(C) Drucken wir nun den neuen Inhalt von »test« aus, dann sieht es fast so aus, als sei unsere Operation gelungen. Aber eben nur fast! Zählen Sie mal nach, wieviele Zeichen ersetzt worden sind: es sind genau 9, also eines weniger als wir eigentlich wollten. Dafür hängen hinten weitere, zum Teil eigenartige Zeichen dran.

(D) Es kommt noch schlimmer: Lassen wir uns nämlich die Länge des veränderten Strings aus »test« ausgeben, dann wird, scheinbar völlig unmotiviert, 42 dafür ausgegeben, obwohl wir uns doch ganz sicher sind, daß wir nur 20 Stück eingegeben haben. Sollte **FillChar** auch den String verlängern? Wenn ja, warum werden dann nicht mehr Zeichen ausgegeben und warum gerade 42?

Das sind Fragen, die einen zum Wahnsinn treiben können, wenn man nicht weiß, was sich da eigentlich abspielt. Wir werden die Antworten dazu finden, aber zunächst sollten Sie sich von der Richtigkeit unserer seltsamen Behauptungen selbst überzeugen:

```
      Program FillTest1; (* P44 *)
      TYPE
        180=String[80];
      VAR
        test:180;
      BEGIN
        ClrScr;Write('Test mit FillChar:'^J^J^M);
(*A*)   test:='12345678901234567890';
        WriteLn(test);
        WriteLn('Laenge des Strings vorher:  ',Length(test),^J^M);
(*B*)   FillChar(test,10,'*');
(*C*)   WriteLn(test);
(*D*)   WriteLn('Laenge des Strings nachher: ',Length(test));
      END.
```

Ganz toll wird es erst, wenn wir bei (A) einen längeren String durch Einfügen einer Zeile wie beispielsweise

```
test:=test+test+test;
```

erzeugen und ihn dann mit **FillChar** behandeln. Eigentlich müßten jetzt insgesamt 60 Zeichen in diesem String enthalten sein, was ja durchaus zulässig sein müßte, weil er vom Typ **String**[80] ist. Doch jetzt ist Turbo wieder konsequent und druckt eisern nur 42 von diesen Zeichen. Auch in der Länge läßt es sich nicht beirren. Es bleibt auch hier bei 42.

Die Erklärung finden wir schnell, wenn wir das Zeichen '*' durch den entsprechenden ASCII-Wert ersetzen. Schauen wir in der Tabelle nach, dann ist dies die Nummer 42(!).

Na, sind Sie dahintergekommen? Es fehlt doch beim Ersetzen mit **FillChar** scheinbar ein Zeichen. Nachdem wir getrost annehmen können, daß die Turbo-Anweisungen einwandfrei arbeiten, werden mit Sicherheit auch durch **FillChar**(test,10,42) zehn Zeichen aufgefüllt. Die Füllung beginnt mit der ersten Adresse der Variablen »test«.

Das heißt aber noch lange nicht, daß dort auch schon das erste Zeichen des Variableninhalts steht.

Was schließen wir messerscharf? Das erste Byte hat eine andere Bedeutung: es gibt nämlich bei einer Stringvariablen immer die Länge des Strings an. Damit erklärt sich auch die maximale Stringlänge mit 255 Zeichen, weil dieses Längenbyte eben nur bis 255 hochgezählt werden kann und eine Zweibyte-Länge nicht vorgesehen ist.

Und jetzt erkennen wir: Das Zeichen '*' wird mit seinem Code 42 an der ersten Stelle des Strings durch **FillChar** abgelegt. Und ab sofort bleibt der Prozedur **Write** nichts anderes übrig, als zu versuchen, 42 Zeichen aus dem String »test« auszugeben. Und die findet sie auch, weil auch die Adressen hinter dem eigentlichen Ende des Strings irgendeinen Bytewert enthalten.

Unser Resümee: Vorsicht ist geboten bei der unüberlegten Anwendung der Prozedur **FillChar**.

Dabei wäre es so praktisch, wenn man mit **FillChar** einen String erzeugen könnte, der aus lauter gleichen Zeichen besteht. Mit einfachen Stringvariablen ist dies aber nicht möglich, wie unsere Versuche gezeigt haben. Sollen wir vor diesem Problem kapitulieren?

14.4.2 Eine Anwendung von FillChar Programm P45: FillChar, Zeigertyp, New, Mark, Release, String

Unser Ziel ist klar: Wir wollen mit Hilfe der Prozedur **FillChar** einen String mit lauter gleichen Zeichen auffüllen. In erweiterten BASIC-Dialekten gibt es einen entsprechenden Befehl, der mit String$(zahl,art) beschrieben werden kann.

Wir wissen nun schon, daß eine Stringvariable nicht so ohne weiteres mit **FillChar** bearbeitet werden kann, weil das erste Byte, auf das der Variablenzeiger gerichtet ist, immer die Länge des Strings enthält. Es bleibt uns kein anderer Ausweg, als uns eine Routine auszudenken, die eben einen Zeiger verwendet, der auf das nächste Byte gerichtet ist. Und da spielt Turbo durchaus mit.

Wenn unsere Variable »test« heißt, dann finden wir mit **Addr**(test) in jedem Fall auch den Anfang der Variablen. Erhöhen wir ihn um eins, dann erhalten wir genau die Adresse, bei der das erste Byte unseres Strings steht, der durch die Variable »test« vertreten wird.

Wir weisen diesen Wert einem Zeiger »dum« zu, der vom gleichen Typ ist (das muß er aber nicht sein) und lassen mit dieser Zeigervariablen die Anweisung **FillChar** laufen.

Um die Länge des Strings auf das gewünschte Maß zu setzen, bauen wir in das erste Byte der Variablen den entsprechenden Wert ein.

Die Prozedur richten wir so ein, daß wir die Parameter »zahl« und »art« aufrufen können:

```
      PROGRAM FillTest2; (* P45 *)
      TYPE
(*A*)   180=String[80];
        p80=^180;
      VAR
(*B*)   test:180;
        dum: p80;
(*C*) PROCEDURE stringart (VAR test:180;zahl,art:Byte);
        BEGIN
(*D*)     dum:=Ptr(Addr(test)+1);
          FillChar(dum^,zahl,art);
          dum:=Ptr(Addr(test));
          FillChar(dum^,1,zahl);
        END;
(*E*) BEGIN  {Hauptprogramm}
        ClrScr;
        WriteLn('Erzeugen eines homogenen Strings mit FillChar:');
        test:='12345678901234567890';
        GotoXY(1,10);
        WriteLn('urspruenglich: ',test,^J^M);
(*F*)   stringart(test,50,220);
(*G*)   WriteLn('neu belegt:    ',test);
        Write('Laenge des Strings: ',Length(test));
      END.
```

Erläuterungen:

(A) Wir legen je einen Datentyp von der Art »skalar« und »Zeiger« mit der Stringlänge 80 an.

(B) Die benötigten Variablen sind einmal »test«, eine gewöhnliche Stringvariable, und »dum«, die Zeigervariable.

(C) Die Prozedur zum Einfügen nennen wir »stringart« und geben ihr den Parameter

»test« als Variable mit, denn nach der Rückkehr ins Hauptprogramm soll »test« ja anders aussehen als vorher.

Die beiden Festwertparameter »zahl« und »art« sind Übernahmeparameter, die ihren Wert nicht ändern. Sie bringen aus dem Hauptprogramm die Anzahl der einzufügenden Zeichen und den entsprechenden ASCII-Wert mit.

(D) Jetzt folgt der wichtigste Teil: Der mit **Ptr** neu eingerichtete Zeiger »dum« wird mit der um eins erhöhten Adresse belegt, welche von der Funktion **Addr** für die ursprüngliche Variable »test« gefunden wurde.

Wir verwenden »dum«, um erstens auf den Anfang der Zeichenkette zu zeigen und mit **FillChar** unsere selbstgewählten Zeichen darüberzulegen. Zweitens, um die Stringlänge einzubauen.

Das war's eigentlich im Prinzip schon. Einfach ist es, wenn man die Zusammenhänge kennt.

(E) Der Rest ist Sache des Hauptprogramms: Zunächst belegen wir die Variable »test« mit einer beliebigen Zeichenfolge und zeigen das Ergebnis auf dem Bildschirm.

(F) Der Aufruf der neuen Prozedur erfolgt wie vereinbart. Diesmal haben wir z.B. keine Variablen verwendet, um Länge und ASCII anzugeben, sondern gleich die Werte selbst. Auch das akzeptiert Turbo.

(G) Der Testausdruck soll uns den neuen String »test« und die Länge ausweisen, die jetzt verändert sein müßte.

Sollte man aber auf die Idee kommen, mehr als die Anzahl Zeichen zu verändern, die der Datentyp zuläßt (in unserem Fall 80), dann kommt es nicht zu einer Fehlermeldung. **FillChar** prüft nämlich nicht nach, ob das Ende einer Variablen erreicht ist, sondern füllt rigoros so viele Zeichen in den Speicher, wie angegeben wurden. Das hat zur Folge, daß eine eventuell dahinterliegende Variable auch gleich mit überschrieben wird. Wer einen Schritt weitergehen will, kann sich dazu eine Absicherung ausdenken.

Will man die Prozedur noch um eine weitere Stufe ausbauen, kann man auch die Position variabel halten, ab der eingefügt wird. Dann wird noch ein weiterer Festwertparameter vom Typ **Integer** z.B. »pos« fällig, der in der »Zeigerzeile« auftritt:

```
(*D*)   dum:=Ptr(Addr(test)+1+pos);
```

Zur Form:

- Zeiger, die durch eine Zuweisung mit **Ptr** eingerichtet werden, brauchen nicht vorher mit **New** oder **GetMem** installiert zu werden, da sich diese Anweisungen nur auf den Heap beziehen.

- Vergessen Sie beim Aufruf des Zeigers das Kotrollzeichen »^« nicht, sonst entstehen unsinnige Ausgaben.

- Warum wir hier mit Zeigern arbeiten, hat seinen Grund darin, daß eine Anweisung wie z.B. **FillChar**(**Addr**(test)+1,zahl,art) nicht akzeptiert wird, weil **FillChar** eine Variablenadresse erwartet.

- Außerdem sollten bei den Prozedurparametern keine Zeiger als Variablenparameter verwendet werden.

14.4.3 Verschiebung eines Speicherbereichs Programm P46: Move, Mem, Zeiger, Midstr, Tausch, Swap

Eine weitere Möglichkeit, Speicheradressen neu zu belegen, realisiert Turbo mit der Prozedur **Move**. Damit lassen sich ganze Speicherblöcke von einem Bereich in einen anderen verschieben. Als Bezugsadressen werden von **Move** wieder die Anfangsadressen von Variablen erwartet.

So verschiebt z.B. **Move**(ursprung,neustart,500) einen Block von 500 Zeichen, gezählt ab der Anfangsadresse der Variablen »ursprung«, in den Bereich, der mit der Anfangsadresse der Variablen »neustart« beginnt.

Um eine einzelne Speicheradresse zu belegen, stellt Turbo das **Array Mem** zur Verfügung, dessen Einsatzmöglichkeiten wir hier nur kurz streifen.

Bearbeiten wir diesmal gleich eine zweiteilige Aufgabe:

– Eine erste Prozedur soll dazu dienen, die Variableninhalte zweier Stringvariablen zu vertauschen. Sie kennen dazu vielleicht aus anderen Programmiersprachen den Befehl SWAP.

  Turbo kennt auch eine Funktion **Swap**. Diese führt aber bei einer Integerzahl ein Vertauschen von Lo- und Hi-Byte durch, ist also hier nicht zu gebrauchen.

– Das zweite Unterprogramm gestalten wir zur Abwechslung mal wieder als Funktion, die uns aus einem String heraus einen beliebigen Teilstring ausgibt. Der entsprechende BASIC-Befehl dazu wäre MID$.

Das Demo-Programm P46.DEM führt Ihnen das vor:

```
      PROGRAM MoveDemo; (* P46.DEM *)
(*A*) TYPE
        l80=String[80];
        p80=^l80;
      VAR
        kette1,kette2,retter:l80;
        l1,l2:Integer;
        dummy:p80;
(*B*) PROCEDURE tausch (VAR kette1,kette2:l80);
      BEGIN
(*C*)   l1:=Length(kette1);l2:=Length(kette2);
        Move(kette1,retter,l1+1);
        Move(kette2,kette1,l2+1);
        Move(retter,kette2,l1+1);
      END;
(*D*) FUNCTION midstr (alt:l80;von,anzahl:Byte):l80;
      BEGIN
(*E*)   New(dummy);dummy^:=alt;
        dummy:=Ptr(Addr(dummy^)+von-1);
(*F*)   {FillChar(dummy^,1,anzahl);}
        Mem[Addr(dummy^)]:=anzahl;
        midstr:=dummy^;
        {Mark(dummy);} Release(dummy);
      END;
      BEGIN
(*G*)   ClrScr;Write(#27,'p');
        WriteLn('Tauschen zweier Variableninhalte mit Move',^J^J^M);
        Write(#27,'q');
        kette1:='abcdefghijklmnopqrstuvwxyz';
        kette2:='*1*2*3*4*5*6*7*8*9';
        WriteLn('urspr. Inhalt von »kette1«: ',
                kette1,' ',Length(kette1));
        WriteLn('urspr. Inhalt von »kette2«: ',
                 kette2,' ',Length(kette2),^J^M);
        Tausch(kette1,kette2);
        WriteLn('neuer Inhalt von »kette1«:  ',
                 kette1,' ',Length(kette1));
        WriteLn('neuer Inhalt von »kette2«:  ',
                 kette2,' ',Length(kette2),^j^m);
        WriteLn('aus kette1 9 Zeichen beginnend mit dem achten:');
```

```
  Write(midstr(kette1,8,9));
END.
```

Erläuterungen:

(A) »kette1« und »kette2« sowie »retter« sind Stringvariablen mit einer maximalen Länge von 80 Zeichen.

Eine Zeigervariable »dummy« verwenden wir wieder als temporäre Variable.

(B) Die Prozedur »tausch« übernimmt die beiden Stringvariablen, deren Inhalte vertauscht werden sollen.

(C) Nachdem die tatsächlichen Längen der beiden Strings festgestellt wurden, erfolgt der Ringtausch über die Variable »retter«, die man auch lokal deklarieren könnte.

(D) Die Funktion »midstr« bekommt den zu bearbeitenden String in der Variablen »alt« vom Hauptprogramm geliefert, das außerdem noch angeben muß, »von« welchem Zeichen an welche »anzahl« an Zeichen in den neuen String übernommen werden soll.

(E) Mit dem aktivierten Zeiger »dummy« zeigen wir auf die Stelle im alten String, die um eins niedriger liegt als die, bei der das Herauslesen beginnen soll. Das ist ein kleiner Trick, mit dem wir ein zusätzliches Byte für die Stringlänge freihalten.

(F) Diese freigehaltene Adresse überschreiben wir mit der gewünschten Anzahl, die entnommen werden soll, und weisen der Funktionsvariablen »midstr« das Ergebnis zu, wobei uns nicht interessiert, welche Zeichen hinter den gewünschten noch stehen.

Wie Sie im Pascal-Text sehen, verwenden wir dazu einen neuen Begriff. Es ist dies **Mem**, ein vordefiniertes **Array**, mit dem man auf die Speicheradressen des Arbeitsspeichers direkt zugreifen kann, allerdings immer nur auf die aktive RAM-Bank. Wir werden dieses Thema später noch einmal aufgreifen. Zunächst genügt es zu wissen, daß wir die Adresse, an der das »dummy« beginnt, direkt mit **Mem**[dummy^] belegen können.

In Kommentarklammern finden Sie auch die Möglichkeit, dies mit der Prozedur **FillChar** zu tun, mit der man natürlich auch ein einziges Byte überweisen kann.

(G) Das Hauptprogramm dient nur der Demonstration. Es gibt die Strings mit den jeweils aktuellen Inhalten sowie die Stringlängen aus. Die Prozeduraufrufe »tausch« und »midstr« unterscheiden sich durch nichts von den übrigen Turbo-Befehlen.

Kapitel

15

Bearbeitung von Dateien

15.1 Grundsätzliches über Turbo-Dateien

Eine Sammlung von Daten, also eine Datei, kann im Arbeitsspeicher des Computers oder dauerhaft auf einem Datenträger, z.B. einer Diskette, einer Festplatte usw. abgelegt und bei Bedarf wieder abgerufen werden.

Wir werden uns mit Turbos Hilfe die wesentlichen Punkte der Dateiverwaltung zu Gemüte führen:

- Erfassen von Daten
- Speichern auf Diskette
- Laden von Diskette in den Arbeitsspeicher
- Sortieren von Daten im Arbeitsspeicher
- Direktzugriff auf einen einzelnen Datensatz, z.B.
  - zur Korrektur
  - zum Löschen
- Weiterführen einer Datei
- Löschen einer Datei
- Datenausgabe auf dem Bildschirm und auf dem Drucker

Als Beispiel für eine sinnvolle Anwendung greifen wir auf die Berechnung von Entfernungen zurück, die wir in den ersten Kapiteln angesprochen haben. Da es lästig ist, ständig die Ortsnamen und ihre geographischen Koordinaten einzugeben, legen wir dazu eine Datei an, so daß wir die einmal erfaßten Daten auf Diskette ablegen können und nur noch neu hinzukommende eintippen müssen. Turbo unterstützt diese wichtige Aufgabe; denn zur Datenverarbeitung ist ja letztlich der Computer erfunden worden.

Wie allgemein üblich, werden die Daten einer Datei über Datenkanäle dorthin geschleust, wo sie verarbeitet werden sollen. Wir kennen bereits solche Kanäle: Beispielsweise führt der Kanal »OUTPUT« zum Bildschirm, der Kanal »INPUT« kommt von der Tastatur, wenn nichts anderes vereinbart wurde. Deswegen spricht man hier auch von den sogenannten Standard-Dateien **Input** und **Output**. Auch die anderen vom Rechnersystem benützten Geräte, die Peripheriegeräte, werden letztlich wie Dateien behandelt, weil sie wie über Kanäle Daten weiterleiten können.

Dazu bietet das Turbo-System drei Möglichkeiten an, die wir an dieser Stelle lediglich erwähnen und in den folgenden Beispielen gründlich durchleuchten wollen: **File Of** (Datentyp), **Text** und **File**.

15.2 Eine Datei vom Typ File Of Datentyp Programm P60.DEM: Erfassen, Bearbeiten, Ausgeben

15.2.1 Arbeiten mit Daten eines bestimmten Typs: File Of Datentyp

Eine Datei, die vom Typ **File Of** ... ist, besteht grundsätzlich aus Komponenten, die alle vom gleichen Typ sind. Das hat den entscheidenden Vorteil, daß beim Speichern auf einem Datenträger durch einfaches Abzählen der Komponenten jederzeit festgestellt werden kann, wo man sich in der Datei bewegt. Erst damit ist ein sogenannter wahlfreier Zugriff auf eine einzelne Komponente der Datei möglich.

Turbo merkt sich die Position innerhalb einer Datei mit Hilfe eines sogenannten Datei-Zeigers, der automatisch mitgeführt wird, sobald in die Datei geschrieben oder aus der Datei gelesen wird. Er rückt sprungweise vom Anfang einer Datenkomponente zur nächsten.

Mit Hilfe der Turbo-Anweisung **Seek** kann man den Zeiger auf eine beliebige Komponente stellen.

Die Funktion **FilePos** ermittelt die momentane Position des Zeigers. Genauer gesagt die Nummer der Komponente, auf deren Anfang er steht.

FileSize (engl.: Dateigröße) stellt die Größe einer Datei fest; genauer wieder: wie viele Komponenten die angesprochene Datei umfaßt.

Nachdem eine Diskette sektorenweise beschrieben wird, werden vor dem Abspeichern auf Diskette erst einmal so viele Daten gesammelt, wie in einen Sektor passen. Das geschieht in einem Puffer. Volle Puffer werden dann übertragen. Um z.B. auch einen nur teilweise gefüllten Puffer (einen letzten Rest) noch zu übertragen, wird die Prozedur **Flush** verwendet. Sie leert gleichzeitig den Puffer und ermöglicht anschließende Leseoperationen.

Wie diese Anweisungen gehandhabt werden, sehen wir gleich in den folgenden Beispielen.

Für die Daten, also die einzelnen Komponenten der Datei, sind alle nur denkbaren Datentypen möglich. Das heißt, Sie können eine Datei aus einzelnen Zeichen (**TYPE Char**), aus Ganzzahlen (**TYPE Integer**), aus Zeichenketten (**TYPE String**), aus Feldern (**TYPE Array**) usw. oder aus selbstdefinierten Datentypen anlegen. Turbo akzeptiert alles, mit Ausnahme von Dateien, als Komponenten von Dateien.

Einzige Bedingung: Innerhalb einer Datei müssen die Komponenten alle vom gleichen Typ sein. Daß man aber trotzdem innerhalb einer Komponente verschiedene Datentypen unterbringen kann, werden wir gleich sehen, wenn wir an die praktische Programmierarbeit gehen.

Was nicht möglich ist: Der Dateizeiger kann nicht in eine Komponente einer Datei hineingestellt werden, sondern immer nur auf deren Anfang. Aber das ist vollkommen ausreichend.

Merke: Dateien vom Typ **File Of** ... müssen gleiche Komponenten haben. Dadurch wird der Zugriff auf jede beliebige Komponente möglich.

15.2.2 Das Hauptprogramm zu P60.DEM

Im folgenden Beispiel werden wir ein Programm mit etlichen Unterprogrammen

(Prozeduren) aufbauen, das mit der gerade besprochenen Datei vom Typ **File Of** arbeitet.

Das Hauptprogramm ist – wie in fast allen Turbo-Programmen, die gut strukturiert sind – sehr kurz:

```
(* P60: Hauptprogramm zu P60.DEM *)
BEGIN
  ClrScr;
  Write(#27,'2','2');
  REPEAT
    FileOfMenu;
  UNTIL MenuPunkt='9';
  Write(#27,'2','0');
END.
```

Was zu geschehen hat, ist offensichtlich:

Wir schalten zunächst auf den deutschen Zeichensatz um und rufen das Menü namens »FileOfMenu«, also eine Prozedur, immer wieder auf, bis der Menüpunkt Nr.9 gewählt wurde. Das bedeutet Programmende, was uns zur Wiederherstellung des Standard-Zeichensatzes bewegt.

Dieses Menü enthält nun eine Reihe von Bearbeitungsmöglichkeiten, wie sie für Dateien üblich sind. Die einzelnen Punkte entsprechen der Aufstellung aus Abschnitt 15.1.

15.2.3 Grobstruktur eines Dateiverwaltungsprogramms Prozedur P47.INC: » FileOfMenu«

Wir gehen jetzt nicht mehr auf die Arbeitsweise eines Menüs ein. Das haben wir schon ausführlich erledigt, sondern beschränken uns auf die Betrachtung der Unterprogrammaufrufe. Doch hier zunächst der Programmtext für unser Menü. Guten Appetit:

```
      PROCEDURE FileOfMenu; (* P47.INC *)
(*A*) BEGIN
        GOTOxy(1,23);
        Write('0 Erfassen  1 Speichern  2 Laden  ');
        WriteLn('3 Sortieren  4 Korrigieren  5 Loeschen');
        Write('6 Erweitern  7 Schirmausgabe  8 Druckerausgabe');
        Write('  9 Beenden  ');
```

```
        ROn;Write('Taste:');ROff; Write(' ');
        Read(KBD,MenuPunkt);
        CASE MenuPunkt OF
(*B*)     '0': Erfassen(1);
          '1': SpeichernFile(2,1);
          '2': LadenFile;
          '3': BubbleSort;
          '4': Korrigieren;
          '5': LoeschMenu;
          '6': Erweitern;
          '7': Ausgabe(0); {Schirm}
          '8': Ausgabe(1); {Drucker}
          '9': Exit;
        END;  {Ende CASE OF}
      END;    {Ende Prozedur}
```

Erläuterungen:

(A) In zwei Menüzeilen wird auf dem Bildschirm dargestellt, welche Möglichkeiten das Programm bietet. Wir arbeiten hier sehr spartanisch, weil es uns im Moment nicht auf eine besondere Form ankommt. »ROn« und »ROff« stehen für zwei Miniprozeduren, die auf reverse Schrift bzw. Normaldarstellung umschalten.

(B) Mit Taste **<0>** wird das Erfassen der Daten gewählt. Der Parameter »1« legt die Nummer der ersten zu erfassenden Daten fest. In diesem Fall beginnt die Aufnahme mit Nummer 1 von vorn. Das bedeutet, daß eine neue Datei angelegt wird.

Mit **<1>** kann abgespeichert werden. Der erste Parameter legt fest, ob die Datei in allen Teilen neu abgelegt werden soll (2), oder ob bereits bestehende Daten erhalten werden sollen (1).

Der zweite Parameter übergibt die Nummer der ersten Komponente, ab welcher die Datei beschrieben wird.

(2,1) heißt hier: Die Datei wird neu angelegt, die Speicherung beginnt mit der ersten Komponente.

<2> erklärt sich von selbst.

<3> löst das Sortieren in alphabetischer Reihenfolge entsprechend ASCII aus.

<4> bietet die Möglichkeit, auf einen bestimmten Datensatz zwecks Korrektur zuzugreifen.

<5> führt zu einem Untermenü, das entweder das Löschen einzelner Datensätze oder der ganzen Datei bewirkt. Dazu werden die beiden Prozeduren »LoeschEinzeln« bzw. »LoeschDatei« benötigt.

<6> fügt an eine Datei weitere Datensätze an.

<7> schreibt die Daten der Datei auf den Schirm.

<8> springt zur gleichen Routine wie **<7>**, aber mit einem anderen Parameter, der die Arbeit mit dem Drucker auslöst.

<9> beendet das Programm.

Jedesmal, wenn ein Punkt aus dem Menü bearbeitet wird, lassen wir die beiden Menüzeilen verschwinden. Das besorgt die kleine Prozedur »TafelFrei«.

Vor jedem Lade- oder Speichervorgang prüfen wir, ob die angesprochene Datei bereits auf Diskette existiert oder nicht. Dazu bauen wir eigens die Prozedur »DateiName« auf.

Damit ist die Gliederung unseres Dateiverwaltungsprogramms komplett. Es läßt sich schematisch so darstellen:

```
(I)   Deklarierungsteil TYPE,VAR - global

(II)  Deklarierungsteil der Prozeduren:
      TafelFrei;          P48
      Erfassen();         P49
      DateiName();        P50
      Ausgabe();          P53
      SpeichernFile();    P51
      LadenFile();        P52
      BubbleSort;         P54
      Korrigieren;        P55
      Erweitern;          P56
      LoeschEinzeln;     P57
      LoeschDatei;       P58
      LoeschMenu;        P59
      FileOfMenu;         P47

(III) Hauptprogramm;      P60
```

Die Klammern hinter dem Bezeichner deuten an, daß es sich hierbei um Prozeduren handelt, die mit Parametern arbeiten. Die Bezeichnung dahinter gibt an, unter welcher Programmteilnummer wir die einzelnen Unterprogramme besprechen. Alle Prozeduren legen wir als Include-Dateien auf Diskette und holen sie erst beim Kompilieren herein.

Falls Sie sich gewundert haben, daß wir die Reihenfolge gegenüber den Punkten im Menü vertauscht haben, sollten Sie sich daran erinnern, daß eine Prozedur nur aufgerufen werden kann, wenn sie auch örtlich vorher im Text deklariert wurde. Da »FileOfMenu« alle anderen aufrufen kann, muß es eben als letzte Prozedur deklariert werden.

Wenn wir nun in den folgenden Abschnitten die benötigten Unterprogramme aufbauen, dann ist es unser Ziel, Sie mit Turbo etwas vertrauter zu machen. Gerade Dateiverwaltungsprogramme sind bis zum Exzeß ausbaufähig. Wir haben nicht die Absicht, diese Richtung einzuschlagen. Was Sie an Komfort bei der Bedienung des fertigen Programmes vermissen, können Sie aber sicher mit Leichtigkeit selbst konstruieren, das Werkzeug dazu wollen wir Ihnen bieten.

Sollten Sie bei dem einen oder anderen Punkt sagen: »Mensch, das geht doch viel einfacher!«, dann sind Sie auf dem richtigen Weg. Probieren Sie aber bitte auch Ihre vermeintlich bessere Version aus. Erst wenn sie einwandfrei läuft, dürfen Sie jubeln!

15.2.4 Reservieren von Menüzeilen Prozedur P48: DelLine

Die Prozedur »TafelFrei« ist schnell erklärt. Ihr Zweck ist das Löschen der beiden Menüzeilen, in denen die zur Verfügung stehenden Programmpunkte per Tastendruck ausgewählt werden.

In den Turbo-Versionen, in denen die Window-Anweisung nicht möglich ist, arbeitet man mit dem **GotoXY**-Befehl. Das wäre auch unter CP/M folgendermaßen möglich:

```
      PROCEDURE TafelFrei; (* P48.INC *)
      BEGIN
(*A*)   GOTOxy(1,23);DelLine;DelLine;
      END;
```

Erläuterungen:

(A) Wir führen den Cursor an die erste der beiden zu löschenden Zeilen und ziehen mit zweimaligem Aufruf von **DelLine** den darunterstehenden Text – also auch die Leerzeilen – nach oben.

Damit ist dieses Feld gelöscht und der Cursor steht am Anfang der 23. Zeile.

15.2.5 Erfassen von Daten Prozedur P49.INC: Record

Wie man Daten erfaßt und sich gegen Fehleingaben absichert, haben wir in den ersten Kapiteln ausreichend behandelt. Damit unser Programm nachher auch läuft, benötigen wir wieder einen Erfassungsteil. Aber damit wir nicht noch einmal Altes aufwärmen, lernen wir bei dieser Gelegenheit einen Datentyp kennen, der sehr flexibel ist: den sogenannten **Record**-Typ.

Records sind nichts anderes als Datensätze. Turbo erlaubt es aber, diese Daten zu untergliedern. Deswegen gehören **Records** zu den strukturierten Datentypen. Das Vorteilhafte daran ist, daß die einzelnen Glieder eines **Records** vollkommen unterschiedliche Datentypen sein können.

Etwas bleibt allerdings konstant: Es ist die Zahl und die Art der Untergliederung und damit die Länge des **Records**. Gleichzeitig ist hiermit aber auch die Bedingung für eine Datei vom Typ **File Of** erfüllt, nämlich die Verwendung von Daten gleichen Typs.

Wie legt man nun einen **Record** an?

Gehen wir von unserem Beispiel aus, daß wir einen Ortsnamen mit einer Länge von 16 Zeichen und zwei geographische Koordinaten mit je einer Länge von maximal 8 Zeichen zulassen wollen, dann läßt sich die Deklarierung der Typen so vornehmen:

```
TYPE
L16=String[16];
L8=String[8];
RecordTyp=Record
            ort:   L16;
            laenge:L8;
            breite:L8;
          END;
FileArray=Array [1..100] Of RecordTyp;
```

Zur Form:

- Die Einleitung beginnt mit dem reservierten Wort **Record**. Danach folgen die Bezeichner für die einzelnen Glieder des **Records**, hinter denen nach einem Doppelpunkt die jeweiligen Einzeltypen angegeben werden müssen.

- Den Abschluß einer **Record**-Definition bildet ein **END** mit einem Strichpunkt dahinter.

- In unserem Beispiel besteht der **Record** aus drei Gliedern, die aus zwei verschiedenen Datentypen gebildet worden sind.

- Selbstverständlich lassen sich solche **Records** auch für die Bildung eines **Arrays** verwenden. Ebenso ist eine Schachtelung von **Records** möglich.

- Unsere Datei wird mit der Variablen »DatFile« geführt und ist vom Typ »FileArray«, wie wir ihn oben beschrieben haben.

Doch zunächst zur Prozedur, die uns das Erfassen der Daten erlaubt. Auch hier haben wir allen Schnickschnack und alle Absicherungen weggelassen und uns nur auf das Wesentliche konzentriert.

Zusätzlich zu den **Record**-Variablen »ort«, »laenge« und »breite« finden Sie weitere Variablen:

– Die Zählvariable »i« (**Integer**), mit der die aktuelle Nummer des einzugebenden **Records** angesprochen wird.

– Die Variable »letzter« (ebenfalls **Integer**), welche die Nummer des letzten Eintrags speichert.

– Eine Variable »dummy«, mit der wir eine Abbruchbedingung schaffen. Sie ist vom gleichen Typ wie die **Record**-Variable »ort« und führt zum Abbruch des Erfassungsteils, wenn ihr Inhalt »X« ist.

– Das **Array** »NavData«, das aus 100 **Records** der beschriebenen Art bestehen kann.

```
(*A*) PROCEDURE Erfassen(nummer:Integer); (* P49.INC *)
      BEGIN
        ClrScr;
(*B*)   i:=Nummer-1;
```

```
(*C*)    dummy:='';
         WHILE dummy<>'X' DO
(*D*)    BEGIN
           i:=i+1;
           WITH NavData[i] DO
           BEGIN
             Write(i:2,'. Ort:     ');
             BufLen:=16;Read(dummy);
(*E*)       IF dummy='X' THEN DelLine ELSE
             BEGIN
(*F*)          ort:=dummy;
               Write(^J^M,'     Laenge:  ');
               BufLen:=8;ReadLn(laenge);
               Write('     Breite: ');
               BufLen:=8;ReadLn(breite);WriteLn;
(*G*)          IF i>letzter THEN letzter:=i;
             END;
           END;
         END;
       END;
```

Erläuterungen:

(A) Im Menü finden Sie den Aufruf der Prozedur »Erfassen(1)«. Damit wird z.B. dem Parameter »nummer« der Wert 1 übergeben. Die Datenerfassung geht also mit dem ersten **Record** los.

(B) Zur Initialisierung löschen wir den Schirm samt Menüzeilen und setzen den Nummernzähler um eins zurück, weil die Erfassungsschleife mit dem Hochzählen der Variablen »i« beginnt.

(C) »dummy« muß geleert werden, sonst findet die Prozedur nach einem weiteren Aufruf sofort die Abbruchbedingung vor, und eine weitere Datenerfassung kann nicht stattfinden.

(D) Solange der Abbruch mit »X« nicht stattgefunden hat, läuft die Erfassungsschleife (eine **WHILE** ... **DO**-Schleife). Da sie mehrere Anweisungen umfaßt, muß sie zwischen **BEGIN/END**-Begrenzer gesetzt werden.

(E) Falls das Dummy die Abbruchbedingung enthält, löschen wir als ordentliche Menschen die unnötig gewordene Zeile, andernfalls (**ELSE**) läuft das Einlesen (**Read**) der Daten so ab, wie wir das schon kennen.

(F) Der Unterschied zu den bisher behandelten Variablen ist, daß wir die einzelnen **Record**-Variablen ansprechen müssen. Und das kann auf zweierlei Art geschehen.

Die erste Möglichkeit haben wir oben vorgestellt:

```
With NavData[i] DO BEGIN ... END;
```

bedeutet in etwa: »Mit den **Record**-Variablen des **Records** Nr.i soll das geschehen, was nun zwischen **BEGIN** und **END** steht«.

Im folgenden Anweisungsblock dürfen aber dann nur die einzelnen **Record**-Variablen angesprochen werden, was wir auch getan haben.

Die zweite Möglichkeit verlangt immer den **Record**-Bezeichner und dahinter nach einem Punkt die einzelne **Record**-Variable. Dafür darf jedoch die **With**-Anweisung nicht auftreten.

Ab (F) müßte der Programmteil dann so aussehen:

```
NavData[i].ort:=dummy;
Write...
BufLen:=8;ReadLn(NavData[i].laenge);
Write...
...         ReadLn(NavData[i].breite);
```

Und nicht vergessen: Die Zeile **With** NavData[i] **DO** muß entfallen.

(G) Wenn der Zähler »i« signalisiert, daß mit dem soeben erfaßten **Record** die Zahl der bereits vorhandenen Datensätze überschritten wurde, muß die Variable »letzter« auf den neuen Wert gebracht werden. Das ist im Falle einer Korrektur notwendig, damit bereits bestehende **Records** nicht abgeschnitten werden.

Zur Form:

- Es ist hier nicht möglich, dem **Record** als Ganzes einen Wert zuzuweisen, also etwa NavData[i]:='Astadt 11.30.20 46.30.10'.

- Werden kürzere Eingaben geliefert, als sie der Datentyp vorsieht, dann bleibt der

vom **Record** reservierte Platz einfach frei. Längere Eingaben würden abgeschnitten, was wir aber mit **BufLen** von vornherein nicht zulassen.

- Die **With**-Anweisungen können ebenfalls geschachtelt auftreten, wenn geschachtelte **Records** dies erfordern. Allerdings ist zunächst nur eine Schachteltiefe von 2 Stufen voreingestellt.

Ein kleines Beispiel für eine Schachtelung in 2 Stufen soll an dieser Stelle genügen. Sie ist dann denkbar, wenn man die geographische Länge noch mit »West« und »Ost« und die Breite mit »Nord und »Süd« unterscheiden will. Schauen Sie sich dazu die folgende Typdeklarierung und einen fiktiven Anweisungsteil an:

```
TYPE
  KoordinatenTyp=Record
                   richtung:String[1];
                   gradzahl:String[8];
                 END;
  RecordTyp=     Record
                   Ort:    String[8];
                   Laenge:KoordinatenTyp;
                   Breite:KoordinatenTyp;
                 END;
VAR NavData:Array [1..100] Of RecordTyp;
BEGIN
  ...
  With NavData[i] DO
  BEGIN
    ReadLn(Ort);
    With laenge DO
    BEGIN
      Read(richtung);
      Read(gradzahl);
    END;
    With breite DO
    BEGIN
      Read(richtung);
      Read(gradzahl);
    END;
  END;
  ...
```

Ob eine Feinstrukturierung mit **With**-Schachtelungen notwendig ist oder nicht, hängt von der Art der Weiterverarbeitung der Daten ab. Falls es wichtig ist, die Richtungsunterscheidungen zu treffen, weil sie als Einzelelemente benötigt werden, bietet sich diese Form auf jeden Fall an.

Als Voreinstellung erlaubt Turbo zunächst zwei ineinandergeschachtelte **With**-Anweisungen. Mit dem Compilerbefehl **{$WX}** kann man die Schachtelungstiefe bis auf 9 Stufen erhöhen: **{$W9}** muß dann vor dem ersten Auftreten einer **With**-Anweisung im Programmtext stehen.

15.2.6 Eröffnung einer Datei Prozedur P50.INC: Assign, Reset, IOresult, $I-, $I+

Um eine Diskettendatei anzusprechen, wird eine Dateivariable benötigt. Wir haben sie in unserem Beispiel »DatFile« genannt. Mit ihrer Hilfe erkennt Turbo, welche Art von Daten in dieser Datei verwaltet werden. Bei uns ist das ein **Array** aus **Records**.

Jede Diskettendatei hat aber auch einen Dateinamen, der erst der Dateivariablen zugewiesen werden muß. Das wird mit der Anweisung **Assign** erledigt.

Das Vorgehen im Programmtext kann dann so aussehen:

- Der Name der Diskettendatei wird in einer Variablen, z.B. »DatName« bereitgestellt. Das kann über einen **Read**-Befehl gehen.

- Die Zuweisung des Dateinamens auf die Dateivariable erfolgt dann mit der Anweisung **Assign**(DatFile,DatName).

  Eine direkte Belegung, etwa mit DatFile:='navorte' ist nicht möglich.

- Nennt man die Diskettendatei z.B. »navorte«, dann wird sie ansprechbar, wenn man sie mit einem **Reset** eröffnet. Auch dies geschieht wieder indirekt, d.h., der Diskettenname selbst wird nicht verwendet, sondern die Dateivariable, der dieser Name zugewiesen worden ist: **Reset**(DatFile).

Das **Reset** (Zurücksetzen) hat seinen Namen daher, weil damit auch gleichzeitig der Dateizeiger, der bei jeder Schreib- oder Leseoperation auf den Anfang der nächsten Komponente springt, auf Null gestellt wird.

Ist eine Datei mit Namen »navorte« vorhanden, dann steht nach einem **Reset** dieser Zeiger am Anfang der ersten Komponente, also auf Null.

Wird keine Datei dieses Namens gefunden, dann bricht Turbo normalerweise mit einer Fehlermeldung ab. Die interne Funktion namens **IOresult** (Ein-/Ausgabe-Ergebnis) liefert dann einen anderen Wert als Null, was die Fehlermeldung auslöst.

Da es ärgerlich ist, wenn das Programm wegen einer nichtgefundenen Datei aussteigt, kann mit Hilfe der Compileranweisung **{$I-}** die Ausgabe der Fehlermeldung unterdrückt werden. Wir überprüfen den **IOresult** selbst und reagieren entsprechend. Zweckmäßigerweise schaltet man aber nach dem **Reset** die Fehlermeldungen für I/O-Operationen mit **{$I+}** wieder ein.

Die Prozedur »DateiName« führt die eben beschriebenen Überprüfungen durch und fragt immer nach der Richtigkeit der gewählten Datei:

```
      (* P50.INC *)
(*A*) PROCEDURE DateiName (VAR DatName:l12;VAR DatFlag:Integer);
      VAR
        ch:Char;
      BEGIN
(*B*)   REPEAT
          TafelFrei;
(*C*)     Write('Name der Datei: ');ClrEoL;
          Read(DatName);
(*D*)     Assign(DatFile,DatName);
(*E*)     {$I-}
          Reset(DatFile);
          {$I+}
(*F*)     IF IOresult=0 THEN
          BEGIN
            Write(' ist bestehende Datei. ');
            DatFlag:=1;
          END
          ELSE
          BEGIN
            Write(' ist neue Datei');
            DatFlag:=0;
          END;
(*G*)     Write(' Korrekt?  J/N');
          Read(KBD,ch);
        UNTIL UpCase(ch)='J';
(*H*)   Close(DatFile);
      END;
```

Erläuterungen:

(A) Mit dem Parameter »DatName« wird der Name der Diskettendatei in das Hauptprogramm zurückgelesen. Der zweite Parameter »DatFlag« ist entweder die Ziffer »1«, wenn die Datei bereits existiert, oder eine »0«, wenn sie unter dem angegebenen Namen nicht gefunden wurde.

(B) Eine **REPEAT**-Schleife läuft so lange, bis (**UNTIL**) eine korrekte Eingabe bestätigt wurde.

(C) Zunächst wird der Dateiname über die Tastatur verlangt. Mit der Anweisung **ClrEOL** löschen wir fehlerhafte Eingaben, falls die Eingabe wiederholt werden muß.

(D) Das ist die Zuweisung des Dateinamens an die Dateivariable.

(E) Vor dem **Reset** schalten wir die Fehlerbehandlung aus, danach wieder ein.

(F) Je nachdem, welchen Wert **IOresult** liefert, geben wir eine Bildschirmmeldung aus, ob es sich um eine bestehende oder eine eventuell neu anzulegende Datei handelt. Entsprechend wird auch das »DatFlag« gesetzt.

(G) Um dem Anwender die Korrektur eines Irrtums zu erlauben, fragen wir erst noch einmal nach, ob er auch tatsächlich die genannte Datei bearbeiten will. Es ist nämlich ärgerlich, wenn eine bereits bestehende Datei beim vermeintlichen Neuanlegen überschrieben wird.

(H) Der Abschlußbefehl nach einer Dateioperation sollte grundsätzlich das Schließen der geöffneten Datei sein. Es könnte sonst zu unangenehmen Erscheinungen kommen.

Auch der **Close**-Befehl verwendet ausschließlich die Dateivariable und nicht etwa den Dateinamen selbst. Das entspricht in BASIC dem Befehl CLOSE#1 oder ähnlichem, wo auch das Schließen über die logische Adresse, z.B. #1, stattfindet und nicht über den Namen.

15.2.7 Speichern einer Datei vom Typ File Of ... Prozedur P51.INC: Reset, Rewrite, Close, Seek, Record

Nach unseren gründlichen Vorbereitungen ist das Abspeichern auf Diskette kein großes Problem mehr. Wie üblich, wird zunächst die gewünschte Datei eröffnet. Aber nun bieten sich zwei Möglichkeiten an:

Reset bereitet die Datei zum Lesen oder Schreiben vor und behält alle bereits vorhandenen Datensätze bei.

Rewrite legt eine neue Datei an und überschreibt eine eventuell unter dem gleichen Namen bestehende unwiederbringlich.

Um uns die Möglichkeit offenzuhalten, bei Bedarf auch an eine bereits bestehende Datei weitere Datensätze anzuhängen, bedienen wir uns einer weiteren Prozedur, die den Dateizeiger verstellen kann.

Während **Reset** den Zeiger auf Null stellt, bewegt **Seek** ihn an die gewünschte Position (auf den Anfang einer Komponente). Dazu gibt man die Dateivariable und eine Integerzahl an.

Wie das gemacht wird, sehen wir im folgenden Programmteil:

```
(*A*) PROCEDURE (* P51.INC *) SpeichernFile
                (spflag,erstneu:Integer);
      BEGIN
(*B*)   DateiName(DatName,DatFlag);
(*C*)   IF spflag=2 THEN Rewrite(DatFile)
(*D*)   ELSE Reset(DatFile);
(*E*)   IF spflag=1 THEN
        Seek(DatFile,FileSize(DatFile));
(*F*)   FOR i:=erstneu TO letzter DO
(*G*)   Write(DatFile,NavData[i]);
(*H*)   Close(DatFile);
      END;
```

Erläuterungen:

(A) Die Speicherprozedur läuft mit zwei Parametern. Der erste ist ein Speicherflag mit den drei vorgesehenen Möglichkeiten 0, 1 und 2, womit die Art der Speicherung geregelt wird.

Außerdem wird mit »erstneu« eine Zahl übergeben, von der ab die zu speichernden Komponenten gezählt werden.

(B) Vorab wird die eben beschriebene Abfrage des Dateinamens und seine Überprüfung durchgeführt.

(C) Wenn das Speicherflag den Wert 2 enthält, dann wird eine neue Datei angelegt und eine bestehende gleichen Namens überschrieben.

(D) Andernfalls wird die Datei lediglich normal geöffnet.

(E) Sollte das Speicherflag den Wert 1 aufweisen, dann bedeutet dies, daß an eine bestehende Datei weitere Daten angefügt werden sollen. Dazu muß der Dateizeiger an das Ende der Datei geführt werden.

Ruft man **FileSize**(DatFile) auf, dann erhält man mit dieser Funktion eine Integerzahl, die die Anzahl der Datensätze enthält (gezählt von eins aufwärts).

Die Prozedur **Seek**(DatFile,n) stellt den Zeiger auf die Komponente mit der Nummer n (von Null an gezählt).

Kombiniert man beide Befehle, dann wird der Zeiger an das Ende der Datei bewegt.

(F) Die eigentliche Speicher-(Schreib-)schleife beginnt jetzt erst: Die erste Datenkomponente hat die Nummer »erstneu«, die letzte wird unter »letzter« geführt.

(G) Die Schreibanweisung muß nun die Dateivariable enthalten, sonst wird die Standard-Datei **Output**, also der Bildschirm, angenommen.

Bei der Arbeit mit **Records** genügt es, wenn der **Record**-Bezeichner angegeben wird. Damit werden auch alle Inhalte der zugehörigen **Record**-Variablen übertragen. Bei der Bildschirmausgabe ist dies jedoch nicht möglich.

Wenn Sie versuchen, mit **Write**(NavData[2]) den zweiten **Record** auf den Bildschirm zu bringen, enttäuscht Sie Turbo mit einer I/O-Fehlermeldung.

(H) Nicht vergessen: **Close** und **Reset** bzw. **Rewrite** gehören immer zusammen zur Behandlung einer Datei. Lediglich bei Verwendung der Standard-Dateien sind diese Eröffnungs- und Schlußzeremonien nicht notwendig (und auch nicht erlaubt).

15.2.8 Laden einer Datei Programmteil P52.INC: Reset, EOF, Close

Entsprechend dem Speichern geht das Hereinladen einer Datei in den Arbeitsspeicher vor sich. Dazu müssen Variablen vorbereitet sein, die die Daten aufnehmen können. In unserem Beispiel sind es natürlich die **Records** des **Array** »NavData«.

Das Einlesen geschieht Nummer für Nummer. Will man alle Datensätze hereinholen,

die zur angesprochenen Datei gehören, dann fragt man die Funktion **EOF** (End of File) ab. Sie ist vom Typ **Boolean** und gibt den Inhalt der Standardvariablen **True** aus, wenn der Zeiger am Ende der Datei angelangt ist.

Und so sieht das in Turbo-Pascal aus:

```
      PROCEDURE LadenFile; (* P52.INC *)
      BEGIN
(*A*)   DateiName(DatName,DatFlag);
        Reset(DatFile);
(*B*)   i:=0;
        WHILE NOT EOF(DatFile) DO
        BEGIN
          i:=i+1;
          Read(DatFile,NavData[i]);
        END;
        Close(DatFile);
(*C*)   letzter:=i;
(*D*)   Ausgabe(0);
      END;
```

Erläuterungen:

(A) Zunächst brauchen wir den Namen der Datei, die wir laden wollen. Dies und die notwendige Überprüfung besorgt unsere Prozedur »DateiName«. Anschließend erfolgt das Öffnen der Datei.

(B) Der allgemeine Zähler »i« wird initialisiert. Über ihn werden mit dem **Read**-Befehl aus der angesprochenen Datei so lange **Records** ausgelesen, bis das Ende der Datei erreicht ist.

Man könnte die Zeile so lesen: »Solange es nicht wahr ist, daß das Ende der Datei erreicht ist, ist folgendes zu tun:...«.

(C) Nach dem Schließen der Datei wird die Nummer des zuletzt gelesenen **Record**s in die Variable »letzter« gerettet.

(D) Zur Kontrolle erfolgt eine Bildschirmausgabe mit einer flexiblen Prozedur namens »Ausgabe«, die wir uns sofort anschauen werden.

15.2.9 Ausgabe von Daten auf beliebige Geräte Programmteil P53.INC: Ausgabetreiber

Eine Ausgabe der eingelesenen Daten auf dem Bildschirm ist für uns kaum mehr ein Problem. Wie aber leitet man die Ausgabe auf den Drucker oder ein anderes Gerät um? Und wie läßt sich eine Routine dafür universell einsetzen? Unser Beispiel soll darauf Antwort geben. Doch vorab sind dazu einige Erklärungen notwendig.

Turbo kennt eine Reihe von Ein- und Ausgaberoutinen und macht uns mit diversen Zeigern den Zugang möglich. Soll z.B. auf die Konsole und damit auf den Bildschirm ein Datensatz ausgegeben werden, dann wird dementsprechend die Prozedur »conout« aufgerufen, die bei einer bestimmten Speicheradresse beginnt, nämlich bei **ConOutPtr**. Und wie man Zeiger abfragt, haben wir bereits gelernt.

Auch für den Drucker ist so eine Einsprungadresse vorgesehen, welche vom Zeiger **LstOutPtr** (von Lister) verwaltet wird.

Außerdem sind noch zwei weitere solche Ausgabezeiger vorhanden, die sich der Benutzer zu eigen machen kann: **AuxOutPtr** (von auxiliary, also Hilfszeiger) und **UsrOutPtr** (von user, Benutzer).

Da eine Zuweisung der Zeiger untereinander möglich ist, kann man hier wunderbar manipulieren:

Wir weisen dem Hilfsgerät **Aux** z.B. den Ausgabezeiger für die Bildschirmausgabe zu, wenn wir etwas auf dem Schirm sehen wollen, oder aber wir weisen ihm den Zeiger für die Druckerausgabe **LstOut** zu, wenn wir unsere Daten gern gedruckt hätten.

Die Ausgaben laufen in jedem Fall über das Hilfsgerät **Aux**, nur der interne Ausgabezeiger hat sich geändert, so daß die Ausgabe auf dem gewünschten Gerät erfolgen kann. Da haben wir also die angestrebte Datenumleitung.

Aber ein wenig aufpassen muß man schon, wenn man die Zeiger verstellt. Man kann nämlich z.B. auch dem **ConOutPtr** den **LstOutPtr** zuweisen. Doch damit ist der ursprüngliche **ConOutPtr** verloren und eine Bildschirmausgabe ist in dieser Form nicht mehr möglich, wenn man den alten Zeiger nicht in eine Variable gerettet hat.

In der Regel ist unter CP/M mit **Aux** die Schnittstelle RDR und PUN vorgesehen. Da wir sie hier nicht benötigen, können wir den **AuxOutPtr** für unsere Zwecke hinbiegen. Sollte aber ein Programm diesen Port benützen, dann muß der Zeiger vorher gesichert und nach der artfremden Benützung wieder installiert werden.

Außerdem ist es zulässig, irgendeinen Zeiger in einen beliebigen freien Adreßbereich zu stellen und dort eine eigene Ausgabe-Routine auf ein beliebiges Gerät zu starten. Solche Programmteile nennt man dann benutzergeschriebene I/O-Treiber.

Eine Anwendung solcher Treiber wäre z.B. die Anpassung an einen bestimmten Druckertyp, wo zuerst bestimmte Zeichen umcodiert werden müssen (Umlaute, Klammern usw.), um in der richtigen Form ausgedruckt zu werden. Probieren Sie da mal ein bißchen herum, aber schauen Sie sich zunächst unsere Ausgabe-Routine an:

```
(*A*) PROCEDURE Ausgabe(geraet:Integer);   (* P53.INC *)
      BEGIN
(*B*)   If geraet=1 THEN AuxOutPtr:=LstOutPtr
(*C*)   ELSE AuxOutPtr:=ConOutPtr;
        ClrScr;
(*D*)   FOR i:=1 TO letzter DO
        BEGIN
          WITH NavData[i] DO
(*E*)     WriteLn(AUX,i:3,'. ',ort:16,'  Laenge: ',
                      laenge:8,'  Breite: ', breite:8);
        END;
      END;
```

Erläuterungen:

(A) Mit dem Parameter »geraet« wird entweder eine 0 für die Bildschirmausgabe oder der Wert 1 für die Druckerausgabe übernommen.

(B) Falls eine Druckerausgabe gewünscht wurde, weisen wir dem Hilfszeiger **AuxOutPtr**, der z.Z. nicht benötigt wird, den Zeiger für die Druckerausgabe zu.

Ebensogut könnten wir auch **UsrOutPtr** verwenden. Das sollte man auch, wenn man sonst mit dem **Aux**-Gerät in Konflikt kommt.

(C) Andernfalls wird der Ausgabezeiger für die Konsole aktiviert.

(D) Mit einer **FOR**-Schleife zählen wir die Ausgaben von der ersten bis zur letzten mit. Die Ausgabe muß jetzt über die einzelnen **Record**-Variablen führen. Das Ansprechen des gesamten **Record**s ist nicht möglich.

(E) Geschrieben (ausgegeben) wird nun immer über das Gerät (eigentlich über die Datei) **Aux**. Je nach Stellung des Zeigers erscheinen unsere Navigationsdaten auf dem Schirm oder auf Papier.

Zur Wiederholung haben wir die Ausgaben formatiert. Ob das wohl der Drucker verkraftet?

Übrigens: Wenn Sie dem Drucker Steuerdaten übersenden wollen, damit er auf Fettschrift umschaltet, einen bestimmten linken Rand einhält oder seine anderweitigen Fähigkeiten zur Schau stellen kann, dann können Sie das natürlich auch von Turbo aus tun.

Nehmen wir als Beispiel das Epson-System her, dann können Sie mit den Escape-Sequenzen genau nach Druckerhandbuch vorgehen. Sie müssen dazu nur den Drucker als Datei **Lst** ansprechen. Hier einige wichtige Schaltungen:

`Write(Lst,Chr(12));`	erzeugt einen Seitenvorschub
`Write(Lst,#27,'E');`	schaltet Fettdruck ein
`Write(Lst,#27,'F'),`	schaltet Fettdruck aus
`Write(Lst,#27,'l',CHR(8));`	setzt linken Rand auf achtes Zeichen
`Write(Lst,#27,'W');`	Breitschrift ein/aus
usw.	

Bei anderen Druckersystemen sind eventuell andere Codierungen vorgesehen. Halten Sie sich dazu an die Anweisungen des Handbuchs.

Anmerkung: Ebenso wie die Ausgaben werden auch die Eingabe-Routinen über Zeiger gesteuert. Die Standardfunktionen ConIn, AuxIn und UsrIn werden über die Zeiger gleichen Namens (z.B. **AuxInPtr**) aufgerufen. Auch sie lassen sich gegenseitig oder anderweitig zuweisen.

Alle diese Treiber-Routinen arbeiten intern ausschließlich mit Daten des Typs **Char**. Das bedeutet aber nichts anderes, als daß Zeichen für Zeichen eingelesen bzw. ausgegeben wird.

15.2.10 Ein Sortierverfahren: Bubble-Sort Prozedur P54.INC: Hilfsvariable, Copy, Vergleich

Es gibt eine ganze Reihe leistungsfähiger Sortierverfahren, die in sehr kurzer Zeit sehr viele Daten nach irgendwelchen Kriterien sortieren. Wir beschränken uns hier auf eine relativ einfache Methode, die aber durchaus flott arbeitet, wenn es darum geht, Daten alphabetisch zu ordnen.

Voraussetzung für fast alle Sortieralgorithmen ist, daß die zu ordnenden Elemente in Feldern (**Arrays**) vorliegen, so daß über den Index gearbeitet werden kann.

Wie sortiert man nun alphanumerische Daten, also solche, die aus Buchstaben und Ziffern bestehen? Ganz einfach, man vergleicht auch Strings mit Hilfe der Turbo-Operatoren »<« (kleiner als), »<=« (kleiner oder gleich), »>« (größer als), »>=« (größer oder gleich) oder »=« (ist gleich).

Dabei geht Turbo nach ASCII vor und gibt entweder den Booleschen Wahrheitswert **True** oder **False** aus.

Beispiel 1: `'A' < 'F'`
ist wahr, weil der Codewert von A mit 65 kleiner ist als der Code von F (70)

Beispiel 2: `'a' < 'F'`
ist falsch, weil das kleine a die Codenummer 97 hat.

Turbo kann genausogut ganze Zeichenketten vergleichen:

Beispiel 3: `'Aalen' >= 'Aachen'`
ist richtig, weil der dritte Buchstabe von Aalen eine höhere Codezahl als der dritte Buchstabe im Wort Aachen hat.

Etwas problematischer wird die Sache mit den Umlauten oder dem 'ß'. Diese Buchstaben rangieren meist als Anhängsel vom Alphabet hinter den übrigen Zeichen und weisen höhere Codewerte auf.

Beispiel 4: `'Ölstadt' < 'Stuttgart'`
liefert daher **False**, obwohl mit unserer Denkweise Stuttgart hinten liegen müßte.

Wir wollen hier nicht näher auf die Behandlung der Umlaute oder ähnliches eingehen, sondern uns ein Sortierverfahren näher anschauen. Wer nicht so recht befriedigt ist, daß die Umlaute falsch eingeordnet werden, darf sich selbst eine geeignete Routine ausdenken. Wir geben dazu weiter unten ein paar Tips.

Das Bubble-Sort-Verfahren, um das es im folgenden geht, hat seinen Namen daher, daß ein höherwertiges Element bei jedem Durchlauf wie eine Luftblase (bubble) im Wasser nach oben gespült wird.

An einem kleinen Beispiel schauen wir uns das mal an. Dazu wählen wir der Einfachheit halber vier kurze Wörter (Strings), die in folgender Weise ungeordnet vorliegen. Allerdings bilden diese vier Zeichenketten ein Feld, so daß sie mit Nummern indiziert sind:

```
Titi      Ulm      Golf     Alb
(1)       (2)      (3)      (4)
```

Jetzt verwenden wir zwei Zähler, X und Y, zum Ansprechen der Komponenten dieses viergliedrigen Feldes:

X ist der Zähler für eine Außenschleife, der vom ersten bis zum vorletzten Element hochgezählt wird.

Mit diesem durch X vertretenen Element werden dann die anderen, rechts von ihm stehenden Elemente Y in einer zweiten inneren Schleife verglichen. Falls das mit X erfaßte Element größer ist, wird es gegen das mit Y erfaßte ausgetauscht. Ist dies nicht der Fall, passiert nichts.

Das klingt kompliziert, löst sich aber in Wohlgefallen auf, wenn wir es mit unseren bereitgestellten Elementen durchführen (der Pfeil ===> bedeutet »Wenn ... dann«):

```
X=1;  Y=2; ===> Titi > Ulm (falsch)
      Y=3; ===> Titi > Golf (wahr)
           ===> Golf    Ulm    Titi    Alb
(1)     (2)     (3)     (4)
      Y=4; ===> Golf > Alb  (wahr)
           ===> Alb     Ulm    Titi    Golf
(1)     (2)     (3)     (4)
```

Damit ist der erste Durchlauf der Innenschleife beendet. Das kleinste Element steht schon mal an der richtigen Stelle.

Im zweiten Durchgang wird X erhöht. Y beginnt immer mit dem rechts davon liegenden (um eins höheren) Index:

```
X=2;  Y=3; ===> Ulm  > Titi (wahr)
           ===> Alb    Titi    Ulm    Golf
                (1)    (2)     (3)    (4)
      Y=4; ===> Titi > Golf (wahr)
           ===> Alb    Golf    Ulm    Titi
                (1)    (2)     (3)    (4)
```

Die innere Schleife wird immer kürzer, weil die links stehenden Elemente nicht mehr berücksichtigt werden müssen, sie haben bereits die richtige Position.

```
X=3;  Y=4; ===>  Ulm  > Titi (wahr)
           ===>  Alb    Golf    Titi    Ulm
                 (1)    (2)     (3)     (4)
```

Fertig! Unsere vier Elemente sind alphabetisch geordnet. Sie sehen, X muß nur bis zum vorletzten Element gezählt werden, weil es dann eventuell mit dem letzten vertauscht wird.

Für unsere Zwecke läßt sich dieses Sortierprogramm in folgender Prozedur unterbringen:

```
      PROCEDURE BubbleSort; (* P54.INC *)
      VAR
(*A*)   hilf:RecordTyp;
        x,y:Integer;
      BEGIN
        ClrScr;x:=0;
(*B*)   REPEAT
          x:=x+1;y:=x;
(*C*)     REPEAT
            y:=y+1;
(*D*)       IF NavData[x].ort>NavData[y].ort THEN
            BEGIN
(*E*)         hilf:=NavData[x];
              NavData[x]:=NavData[y];
              NavData[y]:=hilf;
            END;
          UNTIL y=letzter;
        UNTIL x=letzter-1;
(*F*)   GotoXY(30,13);
        Write('Sortieren beendet!');
      END;
```

Erläuterungen:

(A) Zum Vertauschen brauchen wir eine Hilfsvariable, mit der wir eine der zu vertauschenden Komponente retten. X und Y sind die beiden Schleifenzähler.

(B) Die äußere Schleife wird immer mit der Erhöhung des X-Zählers und dem Initialisieren des Y-Zählers eingeleitet.

(C) Die innere Schleife beginnt immer so, daß der Zähler Y um eins größer ist als der Zähler X.

(D) Zum Vergleich dürfen nicht ganze Records verwendet werden, weil immer nur die Inhalte der **Record**-Variablen miteinander verglichen werden können. Wir wollen die Ortsnamen sortieren und wählen deshalb die entsprechende Variable.

Der Vergleich selbst wird mit dem Operator »>« durchgeführt, der die ASCII-Ordnung berücksichtigt.

(E) **Records** können als Ganzes zugeordnet werden. Wir brauchen also keine Einzelzuweisungen für die **Record**-Variablen aufbauen.

Das Vertauschen läuft über die Hilfsvariable »hilf«, in die wir zunächst einen **Record** retten und an dessen Stelle den auszutauschenden setzen.

(F) Um einen Eindruck zu vermitteln, wie lange das Sortieren dauert, drucken wir als Abschluß eine entsprechende Bemerkung auf den Schirm.

Einige Hinweise zum Ausbau des Sortierverfahrens:

- Sollten die Anfangszeichen der zu sortierenden Wörter auch kleine und große Buchstaben sein, dann wird unsere Routine als erstes die Begriffe mit den großen und dahinter die mit den kleinen Anfangsbuchstaben setzen.

  Also in folgender Form (Beispiel):

```
Aalen
Berlin
Coburg
alt
jung
neu
```

  Dies läßt sich bereinigen, wenn man vor dem Vergleichen den ersten Buchstaben grundsätzlich groß wählt. Austauschen und Vergleichen könnte dann so aussehen:

```
(*D*)
  upx:=UpCase(Copy(NavData[x].ort,1,1))+Copy(NavData[x].ort,2,15);
  upy:=UpCase(Copy(NavData[y].ort,1,1))+Copy(NavData[y].ort,2,15);
```

```
   IF upx>upy THEN
(*E*) ...
```

Die beiden Hilfsvariablen »upx« und »upy« müssen zuvor im **VAR**-Teil deklariert werden:

```
upx,upy:L16;
```

Die Routine verliert dadurch naturgemäß etwas an Tempo, arbeitet aber immer noch recht rasant. Versuchen Sie sich mal daran!

- Soll nach anderen Ordnungen als ASCII sortiert werden, dann ist eben eine Liste anzulegen, nach der jeder String durchsucht wird. So wird man z.B. die Umlaute 'ä', 'ö' und 'ü' jeweils hinter 'a', 'o' oder 'u' einfügen. Die Liste, nach der sortiert wird, würde dann so beginnen:

```
['A','Ä','B'..'O','Ö','P'..'U','Ü','V'..'Z']
```

Eventuelle weitere Zeichen können an jeder Stelle aufgenommen werden.

Arbeitet man statt mit den Characters lieber mit den Codezahlen, dann übernimmt man diese aus der ASCII-Tabelle und fügt entsprechende Zahlen für Sonderzeichen ein:

```
[65,91,66..79,92,80..85,93,86..90]
```

Dazu nützt man die Tatsache aus, daß die Elemente einer Menge ebenfalls in geordneter Reihenfolge vorliegen. Selbstverständlich prüft man nicht alle Zeichen der Strings durch, sondern nur so weit, bis sich eine Unterscheidung ergeben hat.

Doch wir wollen nicht zu weit gehen. Probieren Sie selbst Ihre eigenen Vorstellungen aus, wenn Sie unseren Bubble-Sort verbessern wollen.

15.2.11 Korrigieren eines Datensatzes Prozedur P55.INC: Direktzugriff

Hat man alle Datensätze im Arbeitsspeicher, dann läßt sich ein fehlerhafter Teil durch Abfragen der entsprechenden Nummer und Neubeschreiben der entsprechenden Variablen schnell korrigieren. Und auch das Abspeichern bereitet kaum Programmieraufwand, wenn man die Datei neu überschreibt:

```
      PROCEDURE Korrigieren; (* P55.INC *)
      BEGIN
(*A*)   TafelFrei;
        Write('ab welcher Nummer korrigieren: ');
        Read(nummer);
(*B*)   Erfassen(nummer);
      END;
```

Erläuterungen:

(A) Statt der Menüzeilen erscheint die Abfrage nach der Nummer, die korrigiert werden soll.

(B) Springen wir zur Prozedur »Erfassen«, dann lassen sich auch mehrere Datensätze hintereinander korrigieren bzw. neu anlegen, wenn der letzte erreicht ist. Genau das war der Grund dafür, warum wir »Erfassen« etwas komplexer aufbauen mußten.

Das Abspeichern kann nach Rückkehr in das Hauptmenü erfolgen oder mit einer zusätzlichen Zeile:

```
SpeichernFile(1,nummer);
```

Dabei sorgt die Speicherversion mit dem Flagwert 1 dafür, daß erst ab dem ersten korrigierten Datensatz gespeichert wird. Dieser Direktzugriff auf eine bestimmte Stelle der Datei spart Zeit, besonders bei der Korrektur der hinteren Teile.

15.2.12 Erweitern der Datei Prozedur P56.INC: Flags

Eine Datei vom Typ **File Of** läßt sich ohne weiteres durch Anfügen weiterer Datensätze erweitern. Dazu bieten sich zwei Möglichkeiten an:

– Befinden sich die bereits vorhandenen Datensätze im Arbeitsspeicher, dann erfaßt man die zusätzlich gewünschten mit fortlaufenden Nummern und speichert die gesamte Datei noch einmal neu ab.

  Dies ist sehr unkompliziert, hat aber den Nachteil, daß bei umfangreichen Dateien ein eventuelles Laden und Neuspeichern ziemlich lange dauern kann.

– Eine schnellere Variante beschäftigt sich nur mit den neu hinzugekommenen Daten. Denn der Datenzeiger einer **File Of**-Datei läßt sich mit **Seek** und **FileSize** an das Ende bewegen und ein darauffolgender Schreibbefehl beginnt mit dem Abspeichern der neuen Daten exakt hinter dem Ende der alten.

Wir haben bei der Speicher-Routine »SpeichernFile« diese zweite Möglichkeit schon vorbereitet, indem wir eine Variable als Speicherflag »spflag« benützen und so die Art der Abspeicherung regulieren können.

Blättern Sie noch einmal kurz zurück: Wenn das Speicherflag den Wert 1 enthält, dann wird der Dateizeiger an das Ende der Datei bewegt, bevor der eigentliche Schreibvorgang beginnt.

Die Prozedur für das Erweitern fällt damit sehr kurz aus:

```
(*A*) PROCEDURE  Erweitern; (* P56.INC *)
      BEGIN
(*B*)   nummer:=letzter+1;
        Erfassen(nummer);
(*C*)   SpeichernFile(1,nummer);
      END;
```

Erläuterungen:

(A) Die Prozedur »Erweitern« benötigt keine Parameter.

(B) Mit der Variablen »nummer« übergeben wir der Prozedur »erfassen« den Index des ersten neuen Datensatzes.

(C) Wir bereiten das Abspeichern vor, indem wir das Speicherflag auf den Wert setzen, der in der Speicherroutine das Anfügen durch Setzen des Dateizeigers auslöst. Das ist der erste Parameter. Er erhält den Wert 1.

Das Abspeichern beginnt mit der Nummer, die um eins höher liegt als die Nummer des letzten der vorhandenen Datensätze. Sie wird mit dem zweiten Parameter übergeben.

Wir gehen in dieser Version davon aus, daß diese letzte Nummer bereits bekannt ist, d.h., die Datei wurde bereits einmal in den Arbeitsspeicher geladen. Falls dies unerwünscht ist, legen Sie diese Zeile still.

Es könnte aber nun sein, daß der Benutzer seine Datei blind erweitern will, also ohne sie vorher anzuschauen. Dann muß der Programmierer dafür sorgen, daß zumindest die Nummer des letzten Datensatzes festgestellt wird. Am einfachsten geschieht dies, wenn zu Beginn der Prozedur »Erweitern« die Prozedur »laden_file« aufgerufen wird.

Sie sehen, mit gut vorbereiteten Routinen ist der weitere Ausbau eines Turbo-Programms ein Kinderspiel.

15.2.13 Löschen eines einzelnen Datensatzes Prozedur P57.INC: Record-Zuweisungen

Das Löschen eines einzelnen Records beginnt naturgemäß mit der Abfrage der zu löschenden Nummer.

Mit einer Schleife wird dann Datensatz für Datensatz um eins nach unten gezogen. Die Nummer des letzten Records ist dann um eins kleiner. Der bisherige letzte Datensatz ist aber deswegen noch lange nicht entfernt. Denn wenn wir jetzt die neue, bereinigte Datei mit **Reset** usw. abspeichern und uns einen Bildschirmausdruck ausgeben lassen, erscheint der letzte Satz trotzdem wieder. Lediglich die vorhergehenden wurden überschrieben.

Sie sollten so etwas auf jeden Fall einmal selbst ausprobieren und dann erkennen, daß ein Überschreiben mit **Reset** keine Daten am Ende der Datei löscht. Das bedeutet, daß Sie den Dateizeiger irgendwo in Ihre Datei stellen und dort eine Schreiboperation durchführen können, wobei nur so lange überschrieben wird, wie der Dateizeiger durch Ihre Schreiboperationen bewegt wird.

In unserem Falle ist dies aber unerwünscht und wir behelfen uns mit dem Schreiben einer neuen Datei unter dem gleichen Namen, wobei wir natürlich alle Datensätze von der ersten Nummer an im Arbeitsspeicher halten müssen.

Dazu wurde die Prozedur »SpeichernFile« schon vorbereitet: Wenn das Speicherflag den Wert 2 annimmt, dann erfolgt die Neuanlage einer Datei mit **Rewrite**.

```
      PROCEDURE LoeschEinzeln; (* P57.INC *)
      BEGIN
(*A*)   TafelFrei;
        Write('Welche Nummer loeschen: ');
        Read(nummer);
(*B*)   FOR i:=nummer TO letzter-1 DO
        NavData[i]:=NavData[i+1];
```

```
(*C*)    letzter:=letzter-1;
         SpeichernFile(2,1);
       END;
```

Erläuterungen:

(A) Erfassen der Nummer des zu löschenden Datensatzes.

(B) Die Neuzuweisung der Records braucht nicht über die einzelnen **Record**-Variablen erfolgen, sondern umfaßt jeweils den gesamten **Record**.

(C) Die Anzahl der Datensätze wird um eins reduziert, damit bei der Abspeicherung mit dem auf 2 gesetzten Speicherflag rechtzeitig aufgehört wird.

15.2.14 Löschen einer ganzen Datei Prozedur P58.INC: Erase

Eine Datei wird von der Diskette gelöscht, wenn man den Befehl **Erase** anwendet. Die Prozedur, die wir dazu in unser Programm aufnehmen, ist einfach:

Wir erfassen den Namen der zu löschenden Datei mit der bereits bekannten Prozedur »DateiName« und wenden die **Erase**-Prozedur mit der entsprechend belegten Dateivariablen an.

```
PROCEDURE LoeschDatei; (* P58.INC *)
BEGIN
  DateiName(DatName,DatFlag);
  Erase(DatFile);
END;
```

15.2.15 Ein Löschmenü zur Abrundung Prozedur P59.INC: Sub-Menü

Damit wir aus dem Hauptmenü beim Punkt »Löschen« wieder die Auswahl zwischen dem Löschen eines einzelnen Datensatzes und dem Löschen einer ganzen Diskettendatei treffen können, bauen wir ein Untermenü (Sub-Menü) ein, das auch darauf richtig reagieren sollte, wenn man aus Versehen »Löschen« aufgerufen hat. Es ist unangenehm, wenn man dann zum Löschen gezwungen wird, obwohl man das eigentlich gar nicht vorhatte.

Am besten studieren Sie gleich die Prozedur:

```
      PROCEDURE LoeschMenu; (* P59.INC *)
      BEGIN
(*A*)   TafelFrei;
        Write('0=ganze Datei loeschen  1=einzelne Daten loeschen');
        ROn;Write('Taste:');ROff;Write(' ');
(*B*)   Read(KBD,MenuPunkt);
(*C*)   CASE MenuPunkt OF
          '1': LoeschEinzeln;
          '0': LoeschDatei;
        END; {Ende CASE OF}
(*D*)   Ausgabe(0);
      END;
```

Erläuterungen:

(A) Die Hauptmenütafel wird freigemacht. Es erscheint unser kleines Sub-Menü.

Zur Demonstration haben wir hier noch die Prozeduren »ROn« und »ROff« aufgerufen, die die Bildschirmschrift auf revers bzw. normal schalten. Wir bringen sie im Hauptprogramm am besten gleich zu Anfang unter, damit sie von jeder anderen nachfolgenden Prozedur benützt werden können:

```
(* Revers.INC *)
PROCEDURE ROn; {Reverse Schrift ein}
  BEGIN Write(#27,'p');END;
PROCEDURE ROff; {Reverse Schrift aus}
  BEGIN Write(#27,'q');END;
```

(B) Die Variable »MenuPunkt« aus dem Hauptmenü kann durchaus hier Verwendung finden. Die Inhalte können nicht in Konflikt kommen.

(C) Nur wenn die Tasten **<1>** oder **<0>** gedrückt wurden, läuft eine der Löschprozeduren an.

(D) In jedem Fall lassen wir uns das Endprodukt auf dem Bildschirm darstellen. Nach **<1>** erscheint die berichtigte Datei. Nach **<0>** wird wieder die momentan im Arbeitsspeicher vorhandene Datei auf dem Schirm ausgegeben. Sollte dies die gelöschte Datei sein, dann haben Sie noch die Gelegenheit, sie wieder abzuspeichern, falls Sie sich beim Löschen geirrt haben sollten.

15.2.16 Zusammenfassung File Of-Dateien:

- **File Of**-Dateien bestehen aus lauter gleich langen Datensätzen, die strukturiert sein können. Bei jeder Schreib- oder Leseoperation springt der Datenzeiger auf den Anfang des nächsten Datensatzes.

- Schreiben und Lesen wird mit **Read** bzw. **Write** ausgeführt, wobei dahinter in Klammern zunächst die aktualisierte Dateivariable und danach die Datenvariablen stehen müssen.

- Der Dateizeiger wird mit den Anweisungen **Seek**, **FileSize** und **FilePos** manipuliert.

Sicherheitshalber geben wir Ihnen das Programm zur Bearbeitung einer **File Of**-Datei noch einmal als Ganzes vor, u.a. auch deswegen, damit Sie den Deklarierungsteil komplett vorliegen haben.

Außerdem brauchen wir den Turbo-Text für den nächsten Abschnitt, wo wir die zweite Dateiart vom Typ **Text** besprechen. Dazu übernehmen wir alle Programmteile des eben besprochenen Programms, die keiner Änderung bedürfen. Allerdings ist unser Demo-Programm nicht gegen alle möglichen Fehleingaben abgesichert. Lassen Sie sich dazu etwas einfallen, wenn Ihnen hier etwas auffällt: Beispielsweise bricht das Programm mit einer Fehlermeldung ab, wenn Sie versuchen, eine Datei zu erweitern, die noch nicht auf Diskette angelegt wurde.

15.2.17 Listing für Programm P60 (File Of-Datei-Beispiel)

```
(*************************************************************)
(*          TURBO PASCAL auf dem Schneider CPC 6128          *)
(*             Demo FILE OF Datei, Komplettlisting           *)
(*                 Markt & Technik MT 90455                  *)
(*                   (C) W. Kassera 1987                     *)
(*************************************************************)

      PROGRAM File_Of_Datei_Demo;   (* P60DEMO.KPL *)
      TYPE  {global}
        116=String[16];
        18=String[8];
        112=String[12];
        recordtyp=Record
                    ort:       L16;
                    laenge:    L8;
                    breite:    L8;
```

```
                END;  {Ende Record-Deklarierung}
        filearray=Array [1..100] Of recordtyp;
      VAR  {global}
        navdata:filearray;
        i,nummer,letzter,erstneu,spflag,datflag:Integer;
        datfile:File Of recordtyp;
        datname,dummy:l12;
        menupunkt:Char;
      CONST
        fixrecord:recordtyp=
        (ort:'Greenwich';laenge='00.00.00';breite:'51.30.00');

      (* Revers.INC *)
      PROCEDURE ron;  {Reverse Schrift ein}
        BEGIN Write(#27,'p');END;

      PROCEDURE roff;  {Reverse Schrift aus}
        BEGIN Write(#27,'q');END;

      PROCEDURE tafelfrei;  (* P48.INC *)
      BEGIN
(*A*)   GotoXY(1,23);DelLine;DelLine;
      END;

(*A*) PROCEDURE erfassen(nummer:Integer);  (* P49.INC *)
      BEGIN
        ClrScr;
(*B*)   i:=nummer-1;
(*C*)   dummy:='';
        WHILE dummy<>'X' DO
(*D*)   BEGIN
          i:=i+1;
(*E*)     With navdata[i] DO
          BEGIN
            Write(i:2,'. Ort:    ');
            BufLen:=16;Read(dummy);
(*F*)       If dummy='X' THEN DelLine ELSE
            BEGIN
              ort:=dummy;
              Write(^J^M,'    Laenge:  ');
              BufLen:=8;ReadLn(laenge);
              Write('    Breite: ');
              BufLen:=8;ReadLn(breite);WriteLn;
              IF i>letzter THEN letzter:=i;
            END;
          END;
        END;
      END;
```

```
      (* P50.INC *)

(*A*) PROCEDURE datei_name (VAR datname:l12;VAR datflag:Integer);
      VAR
        ch:Char;
      BEGIN
(*B*)   REPEAT
          tafelfrei;
(*C*)     Write('Name der Datei: ');ClrEol;
          Read(datname);
(*D*)     Assign(datfile,datname);
(*E*)     {$I-}
          Reset(datfile);
          {$I+}
(*F*)     IF IOresult=0 THEN
          BEGIN
            Write(' ist bestehende Datei. ');
            datflag:=1;
          END
          ELSE
          BEGIN
            Write(' ist neue Datei');
            datflag:=0;
          END;
(*G*)     Write(' Korrekt?  J/N: ');
          Read(KPD,ch); Write(UpCase(ch));
        UNTIL UpCase(ch)='J';
(*H*)   Close(datfile);
      END;

(*A*) PROCEDURE ausgabe(geraet:Integer);  (* P53.INC *)
      BEGIN
(*B*)   If geraet=1 THEN AuxOutPtr:=LstOutPtr
(*C*)   ELSE AuxOutPtr:=ConOutPtr;
        ClrScr;
(*D*)   FOR i:=1 TO letzter DO
        BEGIN
          WITH navdate[i] DO
(*E*)     WriteLn(Aux,i:3,'. ',ort:16,'  Laenge: ',
                      laenge:8,'  Breite: ', breite:8);
        END;
      END;

(*A*) PROCEDURE speichern_file;   (* P51.INC *)
      BEGIN
(*B*)   datei_name(datname,datflag);
(*C*)   IF spflag=2 THEN Rewrite(datfile)
(*D*)   ELSE Reset(datfile);
```

```
(*E*)    IF spflag=1 THEN
         Seek(datfile,FileSize(datfile));
(*F*)    FOR i:=erstneu TO letzter DO
(*G*)    Write(datfile, navdata[i]);
(*H*)    Close(datfile);
       END;

       PROCEDURE laden_file;    (* P52.INC *)
       BEGIN
(*A*)    datei_name(datname,datflag);
         Reset(datfile);
(*B*)    i:=0;
         WHILE NOT EoF(datfile) DO
         BEGIN
           i:=i+1;
           Read(datfile,navdata[i]);
         END;
         Close(datfile);
(*C*)    letzter:=i;
(*D*)    ausgabe(0);
       END;

       PROCEDURE bubble_sort;   (* P54.INC *)
       VAR
(*A*)    hilf: recordtyp;
         x,y: Integer;
       BEGIN
         ClrScr; x:=0;
(*B*)    REPEAT
           x:=x+1; y=x;
           REPEAT
             y:=y+1;
             IF navdata[x].ort>navdata[y].ort THEN
             BEGIN
               hilf:=navdata[x];
               navdata[x]:=navdata[y];
               navdata[y]:=hilf;
             END;
           UNTIL y=letzter;
         UNTIL x=letzter-1;
(*F*)    GotoXY(30,13);
         Write('Sortieren beendet!');
       END;

       PROCEDURE korrigieren;   (* P55.INC *)
       BEGIN
(*A*)    tafelfrei;
         Write('ab welcher Nummer korrigieren: ');
```

```
        Read(nummer);
(*B*)   erfassen(nummer);
      END;

(*A*) PROCEDURE erweitern;  (* P56.INC *)
      BEGIN
(*B*)   nummer:=letzter+1;
        erfassen(nummer);
(*C*) {  speichern_file(1,nummer);  }
      END;

      PROCEDURE loesch_einzeln;  (* P57.INC *)
      BEGIN
(*A*)   tafelfrei;
        Write('Welche Nummer loeschen: ');
        Read(nummer);
(*B*)   FOR i:=nummer TO letzter-1 DO
        navdata[i]:=navdata[i+1];
(*C*)   letzter:=letzter-1;
        speichern_file(2,1);
      END;

      PROCEDURE loesch_datei;  (* P58.INC *)
      BEGIN
        datei_name(datname,datflag);
        Erase(datfile);
      END;

      PROCEDURE loesch_menu;  (* P59.INC *)
      BEGIN
(*A*)   tafelfrei;
        Write('0=ganze Datei loeschen   1=einzelne Daten loeschen');
        ron; Write('Taste:');roff; Write(' ');
(*B*)   Read(KBD,menupunkt);
(*C*)   CASE menupunkt OF
          '1': loesch_einzeln;
          '0': loesch_datei;
        END;  {Ende CASE OF}
(*D*)   ausgabe(0);
        END;

      PROCEDURE fileofmenu;  (* P47.INC *)
(*A*) BEGIN
        GotoXY(1,23);
        Write('0 Erfassen  1 Speichern  2 Laden   ');
        WriteLn('3 Sortieren  4 Korrigieren  5 Löschen');
        Write('6 Erweitern  7 Schirmausgabe  8 Druckerausgabe');
        Write('  9 Beenden  ');
```

```
        ron;Write('Taste:');roff; Write(' ');
        Read(KBD,menupunkt);
        CASE menupunkt OF
(*B*)     '0': erfassen(1);
          '1': speichern_file(2,1);
          '2': laden_file;
          '3': bubble_sort;
          '4': korrigieren
          '5': loesch_menu;
          '6': erweitern;
          '7': ausgabe(0);  {Schirm}
          '8': ausgabe(1);  {Drucker}
          '9': Exit;
        END;  {Ende CASE OF}
      END;

      (* P60.PAS HAUPTPROGRAMM zu P60.DEM *)
      BEGIN
        ClrScr;Write(#27,'2','2',#27,'0');
        letzter:=0;
        REPEAT
          fileofmenu;
        UNTIL menupunkt='9';
        Write(#27,'2','0',#27,'1');
      END.
```

15.3 Dateien vom Typ Text

15.3.1 Aufbau einer Text-Datei

Die zweite von Turbo ermöglichte Dateiart unterscheidet sich von dem besprochenen Typ **File Of** dadurch, daß die Datensätze nicht vom gleichen Typ sein müssen. Vielmehr können sie von beliebiger Länge sein und aus beliebigen Zeichen bestehen.

Die Abspeicherung erfolgt in einer sequentiellen Datei vom Typ **Text**, d.h., die Zeichen werden in einer fortlaufenden Kette angelegt. Eine Struktur ist dennoch möglich, weil der Aufbau der Datei in Zeilen erfolgt. Das Trennzeichen zwischen zwei Zeilen ist dabei immer ein CR (Carriage Return = ASCII 13).

Mit Hilfe der Turbo-Befehle **Read** und **Write** lassen sich Zeichenketten ohne eine Zeilenende-Markierung übertragen, während die Anweisungen **ReadLn** und **WriteLn** auf diese Begrenzer angewiesen sind.

Der Nachteil gegenüber der **File Of**-Datei ist, daß man nicht direkt auf einen bestimmten Datensatz zugreifen kann, weil die Operationen, die den Datenzeiger beeinflussen (**Seek**, **FileSize** und **FilePos**) bei **Text**-Dateien nicht wirksam sein können.

Der Vorteil ist eine vereinfachte Handhabung und eine größere Flexibilität bei der Datensicherung, weil die Daten nicht an einen bestimmten Typ gebunden sind.

15.3.2 Beispiel für eine Text-Datei

Da wir im vorhergehenden Abschnitt 15.3 die Bearbeitung einer Datei recht gründlich besprochen haben, beschränken wir uns bei der Ausarbeitung eines Beispiels für **Text**-Dateien auf die Unterschiede gegenüber den **File Of**-Dateien. Sie können daher den Aufbau des Programms P60 bzw. P46 vorläufig unverändert beibehalten.

Im Menü ändern sich nur ein paar Parameter der aufzurufenden Prozeduren. Und in den Prozeduren selbst tut sich nur dann etwas Neues, wenn es sich um das Speichern oder das Laden der Datei dreht.

Gehen wir nun die einzelnen Punkte durch:

- Bei der Deklarierung der Variablen muß die Dateivariable, die wir »DatFile« genannt haben, vom Typ **Text** sein.

- Die beiden Flags »erflag« und »spflag« können wir streichen. Alle anderen Variablen behalten wir für unser Beispiel bei.

- Auch die Prozedur »Erfassen« braucht nicht verändert zu werden. Sie liefert uns in **Record**-Form jeweils die Daten für einen Ort (Ortsnamen und geographische Koordinaten).

- Der Dateiname der Diskettendatei wird genauso wie bei der **File Of**-Datei eingelesen und die Datei wird auf ihre Existenz überprüft. Das Flag »DatFlag«, in welchem wir festhalten, ob es sich um eine bestehende Datei handelt oder nicht, behalten wir bei.

- Die Ausgabe-Routine braucht ebenfalls nicht abgewandelt zu werden, genausowenig wie die Prozeduren Sortieren, Korrigieren, Erweitern und Löschen. Lediglich die daran anschließend aufzurufende Routine »SpeichernFile« besitzt keine Parameter mehr.

- Zum Erweitern einer **Text**-Datei stellt Turbo unter CP/M keine eigene Anweisung bereit. Wir ändern die Erweiterungs-Prozedur entsprechend ab.

15.3.3 Speichern einer Text-Datei Prozedur P61.TXT: WriteLn, Write, Rewrite

Aus den vielen Möglichkeiten, die sich bei der Speicherung (Schreiben auf Diskette) ergeben, haben wir drei ausgewählt. Im folgenden Programmtext geben wir Ihnen Gelegenheit, damit zu experimentieren. Sie brauchen nur die Kommentarklammern entsprechend zu setzen, um eine andere Version auszuprobieren. Hier ist die erste Art aktiv:

```
      PROCEDURE SpeichernFile; (* P61.TXT *)
      BEGIN
(*A*)   DateiName(DatName,DatFlag);
        Rewrite(DatFile);
        FOR i:=1 TO letzter DO
        BEGIN
(*B*)   { 1.Moeglichkeit: jedes Glied in eine Zeile }
          WriteLn(DatFile,NavData[i].ort);
          WriteLn(DatFile,NavData[i].laenge);
          WriteLn(DatFile,NavData[i].breite);
(*C*)   { 2.Moeglichkeit: formatierte Zeile bilden }
        { With NavData[i] DO
          WriteLn(DatFile,ort:16,laenge8,breite:8); }
(*D*)   { 3.Moeglichkeit: unformatierte Zeile }
        { With NavData[i] DO
          WriteLn(DatFile,ort,laenge,breite): }
(*E*)   END;
        Close(DatFile);
      END;
```

Erläuterungen:

(A) Wie gewohnt, prüfen wir zunächst nach, ob die Datei unter dem von uns gewählten Namen schon existiert. Das macht wieder die Prozedur »DateiName«.

Im Gegensatz zu der **File Of**-Datei muß die **Text**-Datei bei jeder Änderung von vorn beschrieben werden. Ein **Reset** würde nur eine bestehende Datei ansprechen. Deshalb wählen wir gleich die Anweisung **Rewrite**, die sowohl alte als auch neue Dateien beschreiben kann.

(B) Die erste Möglichkeit, unsere Texte zu speichern, besteht darin, daß wir jede einzelne Angabe als Zeile ausgeben und damit eine klare Trennung zwischen den einzelnen Datensätzen erzielen. Das besorgt die Anweisung **WriteLn**.

Gegenüber der **File Of**-Version beachten Sie bitte, daß keine strukturierten Variablen angenommen werden:

WriteLn(NavData[i]); funktioniert diesmal nicht!

Erlaubt sind nur skalare Variablen, wie sie die einzelnen Record-Variablen darstellen.

Auf der Diskette sehen unsere Einträge dann z.B. so aus, wenn CR das vereinbarte Trennzeichen zwischen den Zeilen darstellt:

```
AstadtCR10.20.30CR45.22.55CRBdorfCR12.00CR50.22CR...
```

Jede Zeile kann von unterschiedlicher Länge sein. Maximal natürlich nur so lang, wie es der Variablentyp erlaubt.

(C) Die zweite Möglichkeit besteht darin, die Zeilen formatiert abzuspeichern und nach jeweils einem Datensatz, bestehend aus dem Ortsnamen und den beiden geographischen Angaben, ein CR auszugeben. Wir haben dies mit der **With**-Anweisung etwas vereinfacht. Bei (B) wäre dies ebenfalls möglich.

Auf der Diskette ergäbe sich dann folgendes (logisches) Bild:

```
----------Astadt10.20.3045.22.55CR-----------Bdorf12.00---50..
```

Die Striche sollen dabei Leerzeichen darstellen. Dieser Aufbau hat den Sinn, daß wir später (beim Lesen) auch ohne das Trennzeichen an Hand der Längen die richtige Anzahl Zeichen übernehmen können.

Der Befehl

```
WriteLn(DatFile,ort,laenge,breite);
```

bewirkt also das gleiche wie die Befehlsfolge

```
Write(DatFile,ort);
Write(DatFile,laenge);
```

```
Write(DatFile,breite);
Write(DatFile,#13);
```

Anmerkung: #13 steht dabei für das ASCII-Zeichen mit der Nummer 13.

(D) Die dritte – hier allerdings nicht empfehlenswerte Möglichkeit – besteht darin, die drei zusammengehörenden Ortsangaben jeweils als Block hintereinander zu schreiben. Wenn man sich überlegt, wie das auf Diskette ausschaut, dann kann man sich vorstellen, welche Schwierigkeiten es bereitet, beim Hereinladen der Daten wieder Ordnung zu schaffen:

```
Astadt10.20.3045.22.33CRBdorf12.0050.22CRCweiler11.0048.23...
```

Noch unübersichtlicher wäre die Schreibweise mit dem einfachen **Write**. Dann gäbe es überhaupt keine Trennzeichen mehr.

Bei **Text**-Dateien gilt daher die Regel: Alle Einzelkomponenten einer Datei sollten in Zeilenform abgelegt werden.

(E) Nicht vergessen: Auch **Text**-Dateien sollten nach jeder Speicheroperation wieder geschlossen werden.

15.3.4 Laden einer Text-Datei Prozeduren P62.TXT, P63.TXT: EOF, EOLN, $V-, $V+

Wichtig ist beim Hereinladen von Diskette, daß eine **Text**-Datei wieder so eingelesen wird, wie sie geschrieben wurde. Um die wichtigsten Spielregeln kennenzulernen, haben wir wieder drei Möglichkeiten zum Laden (entsprechend unserem Beispiel) aufgenommen. Diesmal ist die letztgenannte aktiv, die anderen sind durch Kommentarklammern stillgelegt worden:

```
      PROCEDURE LadenFile; (* P62.TXT *)
      BEGIN
(*A*)   DateiName(DatName,DatFlag);
        IF DatFlag<>0 THEN
        BEGIN
          Reset(DatFile);
          i:=0;
          ClrScr;
(*B*)     WHILE NOT EOF(DatFile) DO
          BEGIN
```

```
            i:=i+1;
(*C*)     { 1. Lesemoegl. geeignet fuer Speicherversion 1 und 2 }
          {   ReadLn(DatFile,NavData[i].ort);
              ReadLn(DatFile,NavData[i].laenge);
              ReadLn(DatFile,NavData[i].breite); }
(*D*)     { 2. Lesemoegl. geeignet fuer Speicherversion 2 }
          {   With NavData[i] DO
              ReadLn(DatFile,ort,laenge,breite); }
(*E*)     { 3. Lesemoeglichkeit universell geeignet }
              With NavData[i] DO
              BEGIN
                {$V-}
                LiesZeile(ort);
                LiesZeile(laenge);
                LiesZeile(breite);
                {$V+}
              END;
            END;
(*F*)       Close(DatFile);
            letzter:=i;Write(Chr(7));
            Ausgabe(0);
          END;
        END;
```

Erläuterungen:

(A) Nach der Prüfung der Datei auf ihre Existenz lassen wir das Laden nur dann zu, wenn eine bestehende Datei angesprochen wurde, denn sonst bricht das Programm mit einer Fehlermeldung ab.

(B) Solange das Ende einer **Text**-Datei nicht erreicht ist, wird die Einleseschleife abgearbeitet. Dazu muß man wissen, daß auf Diskette das Dateiende mit einem CTRL/Z (^Z) markiert wird. Die Standardfunktion **EOF** (End of File) nimmt den Wert **True** an, wenn beim Lesen auf diese Zeichenfolge gestoßen wird.

(C) **ReadLn** liest in die vorgesehene Variable alle Zeichen bis zum Auftreten eines CR ein, wenn der Variablentyp dies von der Länge her zuläßt. Haben wir mit der ersten Speichermöglichkeit die Diskette beschrieben, dann sehen die Inhalte der **Record**-Variablen (und nur solche dürfen hier verwendet werden) so aus:

```
ort[1]='Astadt'; laenge[1]='10.20.30';  breite[1]='45.22.55';
ort[2]='Bdorf';laenge[2]='12.00';breite[2]='50.22'
```

usw.

Haben Sie dagegen mit der zweiten Version gespeichert, dann werden bei den Orten jeweils 16 Zeichen eingelesen, bei den Koordinaten jeweils 8, weil linksbündig mit Leerstellen aufgefüllt wurde.

Böse sieht es dagegen aus, wenn Sie mit der dritten Art Ihre Text-Datei angelegt haben, weil mit einem **ReadLn** entweder bis zum nächsten CR oder bis zum Erreichen der maximalen Stringlänge eingelesen wird:

```
ort[1]='Astadt10.20.3045'; laenge[1]='22.55' (CR);
breite[1]='Bdorf12.0050.22CR'; ort[2]='^Z ?...
```

Was ist da bloß geschehen?

Für die Ortsnamen wurden korrekt 16 Zeichen eingelesen und für die Koordinaten jeweils 8, außer wenn ein CR auftrat. Andererseits wird ein CR als ASCII-Zeichen (Nummer 13) ohne weiteres in einen String aufgenommen, wenn er gerade vorliegt.

Auch das Zeichen ^Z für Datei-Ende wird eingelesen und damit ist unsere Abbruchbedingung ausgeschaltet. Das Programm versucht aber weiterhin von Diskette zu lesen, obwohl dort gar nichts (sinnvolles) mehr steht.

Also Vorsicht: Das Lesen muß zum Speichern passen!

(D) Für die zweite Speicherversion ist der entsprechende **ReadLn**-Befehl möglich. Er entspricht der Befehlsfolge

```
Read(DatFile,ort);
Read(DatFile,laenge);
Read(DatFile;breite);
ReadLn(DatFile);
```

Er operiert aber nur dann sinnvoll, wenn die Daten formatiert gespeichert wurden. Und zwar muß das Speicherformat jeweils dem Variablentyp angepaßt worden sein. Beispielsweise mußten die Orte mit einer Länge von 16 Zeichen abgelegt werden, weil sie vom Typ **String**[16] sind.

Sollten Sie versuchen, mit dieser Version Daten zu lesen, die in einer anderen Form abgespeichert wurden, dann passiert Ähnliches wie unter (C).

(E) Eine der Text-Datei angemessene Lesart ist die folgende:

Man liest Zeichen für Zeichen ein, bis das Ende einer Zeichenfolge erreicht ist, was durch das Auftreten eines CR bemerkt wird.

In Pascal erstellt man sich dazu eine »LiesZeile«-Prozedur. Sie baut in einem Variablenparameter »zeile« den einzulesenden String Zeichen für Zeichen auf und gibt ihn an die aufrufende Prozedur »LadenFile« zurück. Dazu verwenden wir die Funktion **EOLN**, die immer dann den Wert **True** annimmt, wenn ein CR gefunden wurde, und lassen eine **REPEAT**-Schleife so lange lesen, bis eben dies eintritt. Ein weiterer Kommentar erübrigt sich wohl:

```
PROCEDURE LiesZeile (VAR zeile:l16); (* P63.TXT *)
VAR
  ch:Char;
BEGIN
   { 3.Lesemglk. universell liest Zeichen fuer Zeichen }
   zeile:='';
   REPEAT
     Read(DatFile,ch);
     zeile:=zeile+ch;
   UNTIL EOLN(DatFile);
   ReadLn(DatFile);
END;
```

Einen Haken hat die Geschichte aber doch: Bei den Parameterübergaben stimmen die Längen der Stringtypen nicht immer überein. Wir haben zwar in der Prozedur »LiesZeile« den längstmöglichen gewählt. Doch Turbo überprüft streng und meldet einen Fehler, wenn wir Koordinaten vom Typ **String**[8] übernehmen wollen.

Dennoch ist dies mit Turbo möglich! Wir müssen dazu nur den Compiler anders einstellen:

Mit **{$V-}** weisen wir ihn nämlich an, bei der Typüberprüfung der Parameter doch nicht ganz so streng zu sein und unterschiedliche Stringlängen zuzulassen.

Wollen wir wieder strenge Pascal-Disziplin walten lassen, dann muß der Compilerbefehl **{$V+}** ausgegeben werden, der den voreingestellten Zustand wiederherstellt.

(F) Nach dem Schließen der Datei geben wir einen Glockenton als Abschluß der Leseroutine aus. Damit können Sie überprüfen, wie schnell auch das Einlesen und Aufarbeiten einzelner Zeichen vor sich geht. Es dauert nämlich auch nicht

viel länger als das Einlesen ganzer Strings, denn diese müssen ja auch irgendwann einmal aus Einzelzeichen zusammengesetzt werden.

Eine Bildschirmausgabe rundet das Einlesen ab, damit wir sofort überprüfen können, was unser Programm da hereingeschaufelt hat.

15.3.5 Erweitern einer Text-Datei Prozedur P64.TXT

Der Dateizeiger läßt sich in **Text**-Dateien nicht positionieren, und unter CP/M kann man ihn mit Append auch nicht an das Ende der Datei stellen. Damit wird ein erneutes Abspeichern der gesamten Datei notwendig, wenn ein zusätzlicher Textteil an eine bereits bestehende Datei angehängt werden soll.

Benennen wir die abgewandelte Prozedur mit »ErweiternText«:

```
      PROCEDURE ErweiternText; (* P64.TXT *)
(*A*) BEGIN
        nummer:=letzter+1;
        Erfassen(nummer);
(*B*)   SpeichernFile;
      END;
```

Erläuterungen:

(A) Da alle Einträge durchnumeriert sind, erhöhen wir die letzte Nummer um eins und gehen mit dieser zum Erfassen der zusätzlichen Daten.

(B) Nachdem alle gewünschten Eingaben abgeschlossen sind, speichern wir das gesamte File noch einmal ab.

Zusammenfassung Text-Dateien:

- **Text**-Dateien werden in Zeilenform angelegt. **Read** und **Write** lesen bzw. schreiben eine bestimmte Anzahl Zeichen.

- **ReadLn** liest höchstens bis zum Auftreten eines CR, springt aber auf jeden Fall auf den Anfang der nächsten Zeile.

- **WriteLn** schreibt eine Zeichenfolge und fügt als Abschluß ein CR hinzu.

- **ReadLn** und **WriteLn** können auch allein für sich verwendet werden, um den nächsten Zeilenanfang zu finden bzw. um ein Zeilenende zu markieren.

- **EOLN** gibt **True** aus, wenn das Zeilenende erreicht ist. Das Textende wird mit einem ^Z markiert. EOF reagiert darauf mit **True**.

  Anmerkung: EOLN und **EOF** sind sogenannte boolesche Funktionen, die als Ergebnis nur wahr (**True**) oder falsch (**False**) kennen. Mit der **IF**-Anweisung werden diese Wahrheitswerte überprüft.

- Als **Text**-Dateien lassen sich auch andere Diskettendateien ansprechen, z.B. auch die im vorigen Abschnitt besprochene **File Of**-Datentyp-Datei. Umgekehrt ist das aber nicht möglich.

  Achtung: Die externen Geräte wie Konsole (**CON**), Terminal (**TRM**), Tastatur (**KBD**), Drucker (**LST**), alternative Schnittstellen (**Aux**) und benutzerdefinierte Schnittstellen (**Usr**) werden wie **Text**-Dateien behandelt. Dazu werden die vordefinierten Dateinamen (in Klammern) zum Ansprechen dieser Geräte verwendet. Die vorbereitenden Prozeduren **Assign**, **Reset** oder **Rewrite**, sowie das abschließende **Close** werden damit eingespart und sind auch gar nicht mehr zulässig.

15.4 Dateien vom Typ File (typlose Dateien)

Turbo unterstützt die Verwaltung einer dritten Dateiart, die keinen eigenen Typ darstellt, sondern von sehr allgemeiner Art ist und deswegen schlicht mit **File** bezeichnet wird. Mit diesem Typ **File** lassen sich auch alle anderen typisierten Dateien ansprechen, jedoch ist eine derartige Eröffnung nur dann sinnvoll, wenn die Ein- und Ausgaberoutinen dazu passen oder aber, wenn man nur Operationen durchführen will, die nicht die Struktur der Datei betreffen, also z.B. **Rename**, **Erase**, **Execute**, **Chain** usw. (siehe dort). Insbesondere aber sollten alle Dateien, die nicht von einem Turbo-Programm angelegt wurden, mit Hilfe einer typlosen Datei **File** eingelesen werden.

15.4.1 Lesen und Schreiben in Blöcken

File-Dateien werden in ganzen Blöcken zu je 128 Zeichen geschrieben und gelesen. Dafür gibt es eigene Turbo-Anweisungen.

Dabei ist es beim Lesen gleichgültig, ob auf Diskette eine **File Of**- oder eine **Text**-Datei vorliegt. Pro Leseschritt werden immer 128 Zeichen auf einmal hereingeholt.

Beim Schreiben dagegen muß man etwas vorsichtiger sein, weil sonst bestehende Strukturen eventuell durch die Blockstruktur überlagert werden können.

Die schon erläuterten Turbo-Anweisungen **Seek**, **FilePos** und **FileSize** sind auch bei diesen typlosen Dateien anwendbar. Allerdings ist die Einheit für den Dateizeiger hier immer ein ganzer Block.

Im Kapitel 17 behandeln wir ein etwas anspruchsvolleres Problem, bei dem mit typlosen Dateien gearbeitet werden muß. An dieser Stelle begnügen wir uns mit einem Demonstrationsbeispiel, das die notwendigen Elemente vorstellt.

15.4.2 Demo-Programm für den Umgang mit typlosen Dateien Programm P65Block.DEM: Hauptprogramm

Diesmal verzichten wir auf ein Menü und lassen der Einfachheit halber die einzelnen Prozeduren der Reihe nach ablaufen. Damit Sie aber auch gleichzeitig einen Einblick über das Arbeitstempo Ihres Rechners bekommen, stoppen wir nach jeder Prozedur den Programmlauf und geben ihn erst nach einem Tastendruck wieder frei.

Die zu besprechenden Prozeduren binden wir wieder als Include-Files ein, wie wir das in den vorhergehenden Kapiteln schon gesehen haben. Damit wird das Hauptprogramm wieder recht übersichtlich.

```
      PROGRAM Blockdatei; (* P65Block.DEM *)
      TYPE
(*A*)   ZeigerTyp=^Char;
      VAR
        i,j:Integer;
        Puffer:Array [1..24,1..80] Of Char;
        Blockdatei:File;
(*B*) {$I P66.FIL}        {Taste}
      {$I P67.FIL}        {PufferVoll}
      {$I P68.FIL}        {PufferLesen}
      {$I P69.FIL}        {Pufferspeichern}
      {$I P70.FIL}        {BlockDateiLaden}

      PROCEDURE Pufferleer;
      BEGIN
        FOR i:=1 TO 24 DO
```

```
        FOR j:=1 TO 80 DO
        Puffer[i,j]:='0';
      END;

      BEGIN
(*C*)   ClrScr;
        Write(#27,'0');
(*D*)   PufferVoll;
        Taste;
        PufferLesen;
        Taste;
(*E*)   Blockspeichern;
        Taste;
        Pufferleer;
(*F*)   BlockDateiLaden;
        Taste;
        Write(#27,'1');
      END.
```

Erläuterungen:

(A) Die Blockprozeduren arbeiten mit einem Zeiger, dessen Typ wir an dieser Stelle schon global deklarieren können. Das erspart uns die mehrfache Deklarierung in jeder Prozedur.

»i« und »j« sind Zählvariablen.

»Puffer« ist ein zweidimensionales **Array**, das einen Inhalt von 24 Bildschirmzeilen aufnehmen kann.

Mit der Variablen »Blockdatei« bearbeiten wir das typlose **File**.

(B) Die Include-Dateien besprechen wir einzeln.

(C) Der Hauptteil beginnt mit Schirm löschen und Ausschalten der Drive-Anzeige.

(D) Zunächst wird eine Prozedur aufgerufen, die den Puffer mit Zeichen füllt. Anschließend wird dieser Puffer ausgelesen und auf dem Bildschirm dargestellt.

(E) »Blockspeichern« legt eine **File**-Datei an und überspielt den Pufferinhalt auf Diskette.

(F) Zur Kontrolle laden wir die eben angelegte Datei wieder herein und geben sie auf dem Schirm wieder aus. Das erledigt »BlockDateiLaden«.

15.4.3 Prozedur P66.FIL: KeyPressed

Zwischen den einzelnen Aufrufen finden Sie immer wieder die Prozedur »Taste«, unseren Menü-Ersatz:

```
PROCEDURE Taste; (* P66.FIL *)
BEGIN
  Write(^J^M,'Taste tippen!');
  REPEAT Until KeyPressed;
END;
```

In einer neuen Zeile erfolgt die Aufforderung zum Betätigen einer Taste, damit das Programm weiterlaufen kann.

15.4.4 Prozeduren P67.FIL und P68.FIL: Puffer füllen und kontrollieren

Wir geben die Erläuterungen diesmal gleich voraus:

(A) Um die Daten in Blöcken zu je 128 Zeichen verarbeiten zu können, müssen sie erst einmal in einen zusammenhängenden Speicherbereich gebracht werden. Wir besorgen das in der Form, daß wir in einer Hauptschleife 24 »Zeilen« mit je 80 gleichen Zeichen erzeugen.

(B) Um die Zeilen später auseinanderhalten zu können, verwenden wir die Codenummern für die Großbuchstaben, beginnend mit »A«.

```
        PROCEDURE PufferVoll; (* P67.FIL *)
        BEGIN
(*A*)     FOR i:=1 TO 24 DO
          FOR j:=1 TO 80 DO
          BEGIN
(*B*)       Puffer[i,j]:=Chr(i+65);
          END;
          Write('Puffer ist eingelesen!');
        END;
```

Auch die nächste Prozedur ist so einfach, daß wir uns längere Erklärungen ersparen können. Sie liest zur Kontrolle Zeichen für Zeichen aus dem gefüllten Pufferbereich. Man könnte dies auch in einer einzigen **FOR**-Schleife (von 1 bis 1920) erledigen, weil die Daten in einer einzigen Schlange angelegt sind.

```
      PROCEDURE PufferLesen; (* P68.FIL *)
      BEGIN
        ClrScr;
        WriteLn('Daten aus dem Puffer:');
(*A*)   FOR i:=1 TO 24 DO
        FOR j:=1 TO 80 DO
        Write(Puffer[i,j]);
      END;
```

15.4.5 Speichern einer Blockdatei Prozedur P69.FIL: Assign, Rewrite, BlockWrite

Dateien vom Typ **File** müssen wie andere Diskettendateien auch mit **Assign** vorbereitet werden. **Rewrite** legt eine neue Datei an oder überschreibt eine gleichnamige bestehende.

Der Hauptbefehl heißt diesmal **BlockWrite**, hinter dem die anzusprechende Dateivariable angegeben werden muß, sowie die Puffer-Variable, aus der die einzelnen Blöcke zur Übertragung herausgeholt werden sollen. Außerdem ist anzugeben, wieviele Blöcke übertragen werden sollen.

Ein Beispiel: Nehmen wir an, hinter der Speicheradresse »pointer^« stehen die Daten, von denen wir 1280 Zeichen aufnehmen wollen. Dann macht das 10 Blöcke zu je 128 Zeichen. Der Schreibbefehl dazu lautet:

```
BlockWrite(Blockdatei,pointer^,10);
```

Dabei setzen wir voraus, daß mit »Blockdatei« eine geöffnete Datei angesprochen wird.

Als Programmierer müssen Sie darauf achten, daß die Variable auch tatsächlich die zu übertragenden Zeichen gespeichert hat. Dem Befehl **BlockWrite** ist es egal, welche Zeichen hinter der Anfangsadresse der Variablen stehen. Er überträgt auch die Zeichen, die Sie gar nicht bestellt haben, falls der Block über den angesprochenen Variablenbereich hinausreicht.

Sollten Sie aber die Absicht haben, 1300 Zeichen an die Floppy zu senden, dann müssen

wir Sie enttäuschen: es sind nur Vielfache von 128 möglich. Sie müßten dann 11 Blöcke wählen, denn halbe oder zehntel Blöcke gibt es eben nicht.

Nun zu unserem Beispiel:

```
      PROCEDURE Blockspeichern; (* P69.FIL *)
      VAR
        pointer:ZeigerTyp;
      BEGIN
(*A*)   pointer:=Ptr(Addr(Puffer[1,1]));
(*B*)   Assign(Blockdatei,'Block.dem');
        Rewrite(Blockdatei);
(*C*)   BlockWrite(Blockdatei,pointer^,15);
(*D*)   Close(Blockdatei);
        ClrScr;Write('Blockdatei angelegt.');
      END;
```

Erläuterungen:

(A) Wir haben uns ja vorgenommen, mal wieder mit Zeigervariablen zu arbeiten und weisen »pointer« die Anfangsadresse unseres Pufferbereichs zu, also die Stelle, an der das erste Zeichen des **Arrays** steht.

(B) Die Diskettendatei, auf die wir unsere Beispieldaten speichern wollen, nennen wir »Block.dem« und öffnen sie Turbo-gerecht.

(C) **BlockWrite** holt sich nun 15 Blöcke aus dem Puffer. Das macht genau 1920 Zeichen, die auf Diskette übertragen werden.

(D) Nach der Bearbeitung der Datei wird diese wieder geschlossen. Und wir geben das Ende der Operation bekannt.

Zur Form:

- Achten Sie darauf, daß die Zeigervariablen »pointer^« mit dem CTRL-Zeichen dahinter verwendet wird, andernfalls greifen Sie mit »pointer« in Turbos Programmvariablenbereich.

- Wenn man einer Zeigervariable eine bestimmte Adresse zuweist, ist ein vorheriges Anlegen mit **New** oder **GetMem** überflüssig.

- Die Verwendung von Zeigervariablen ist in unserem Demo-Programm nicht obligatorisch. Selbstverständlich können Sie auch mit **BlockRead** und **BlockWrite** auf ein gewöhnliches **Array** zugreifen. Wir wollten hier nur den Umgang mit den Zeigervariablen üben.

  Übrigens: Die Zeigervariable, die wir deklariert haben ist »nur« vom Typ **Char**, was aber keine Rolle spielt, da wir es in unserem Fall nur auf die Anfangsadresse abgesehen haben.

15.4.6 Lesen einer Blockdatei von Diskette Prozedur P70.FIL: Assign, Reset, BlockRead

Der Aufbau der Lade-Prozedur, die wir »BlockDateiLaden« genannt haben, entspricht in etwa der Form des Speicherteils.

Bei den Block-Befehlen ist ein weiterer Parameter möglich, der nach der Übertragungsoperation die Anzahl der tatsächlich übertragenen Blöcke enthält. Wir setzen ihn hier zur Überprüfung ein, wenn das Lesen abgeschlossen ist. Die entsprechende Variable nennen wir einfach »zahl«.

```
        PROCEDURE BlockDateiLaden; (* P70.FIL *)
        VAR
          zahl:Integer;
          zeiger:ZeigerTyp;
        BEGIN
(*A*)     WriteLn(#27,'E','Blockdatei wird geladen');
(*B*)     zeiger:=Ptr(Addr(Puffer[1,1]));
          Assign(Blockdatei,'Block.dem');
(*C*)     Reset(Blockdatei);
(*D*)     BlockRead(Blockdatei,zeiger^,15,zahl);
          ClrScr;
(*E*)     PufferLesen;
        END;
```

Erläuterungen:

(A) Zur Abwechslung löschen wir den Schirm mal mit der ESC-Steuerung »E«, die den Cursor nicht bewegt.

(B) Einer Zeigervariablen wird die Anfangsadresse des Pufferbereichs zugewiesen.

(C) Die Eröffnungsanweisungen müssen diesmal **Reset** enthalten, weil wir auf eine bereits bestehende Datei zugreifen wollen.

(D) Mit 15 Blöcken holen wir wieder den gesamten Inhalt der Datei herein, den wir vorher draufgeschrieben haben. Probieren Sie hier mal andere Blockzahlen aus, auch höhere!

(E) Zur Kontrolle lassen wir uns gleich den Inhalt des Puffers auf den Bildschirm malen. »PufferLesen« können wir dazu ohne weiteres aufrufen, weil es örtlich vorher deklariert ist.

Wenn Sie nun alle XX.FIL-Dateien auf Diskette haben, können Sie das Demo-Programm P65Block.DEM starten.

Zusammenfassung typlose Datei File:

- Dateien vom Typ **File** werden wie jede andere Datei mit **Assign**, **Reset** oder **Rewrite** eröffnet und mit **Close** geschlossen.

- Das Schreiben mit **BlockWrite** legt aus einem Variableninhalt die gewünschte Anzahl Blöcke zu je 128 Zeichen auf Diskette oder eine andere angesprochene Datei.

- **BlockRead** liest entsprechend eine Anzahl Blöcke in einen Speicherbereich ein, der durch eine Variable ausreichender Mächtigkeit reserviert ist. Zeigervariablen für den Heap sind dafür ideal. Ein vierter Kontrollparameter gibt die Anzahl der tatsächlich übertragenen Blöcke aus.

Kapitel

16

Rechnen mit Mengen

Es kann nicht Aufgabe dieses Buches sein, Unterricht in Mengenlehre zu erteilen. Aber da Turbo auch auf diesem Gebiet einigermaßen brauchbar ist, befassen wir uns natürlich mit den gebotenen Möglichkeiten und geben Ihnen dazu auch einen kleinen Einblick in die Behandlung von Mengen, so wie die Mathematik es vorsieht.

16.1 Der Mengenbegriff in Turbo

In der Mathematik lassen sich beliebige Elemente zu einer Menge zusammenfassen. Dabei werden endliche Mengen (mit einer begrenzten Anzahl von Elementen) und nichtendliche Mengen unterschieden. Zahlenmengen wie Q (die Menge aller rationalen Zahlen), R (Menge aller reellen Zahlen) oder C (Menge aller komplexen Zahlen) sind die geläufigsten Vertreter der letztgenannten Art.

Turbo ist hier starken Einschränkungen unterworfen:

- Es werden nur endliche Mengen mit höchstens 256 Elementen verwaltet.

- Alle Elemente einer Menge müssen vom gleichen Typ sein.

- Strukturierte Elemente wie **Records** oder **Arrays** sind nicht gestattet.

Das kommt daher, weil intern jedes Element lediglich mit einem Bit erfaßt wird, so daß aus der geordneten Aufzählung die einzelnen Elemente identifiziert werden können. Das erste Element einer Menge erhält gemäß üblicher Computerlogik die Nummer null, so daß die größtmögliche Ordnungszahl eines Elements 255 sein kann.

Wir haben in den vorhergegangenen Kapiteln die Schreibweise für eine Mengendeklarierung schon kennengelernt. Es ist der Typ **Set Of** ...

In diesem Kapitel wollen wir nun etwas näher auf die Operationen mit Mengen eingehen, ohne uns aber allzuweit in die (nicht immer geliebte) Mengenlehre hineinzusteigern.

16.2 Mengentyp, Mengenvariable, Mengenkonstante

Zur Definition einer Menge wählt man einen Bezeichner aus und weist ihm den gewünschten Typ zu.

Beispiel 1: `MengenTyp=Set Of 0..255;`

Hier werden alle Integerzahlen von 0 bis 255 in einer Menge zusammengefaßt. Eine Einzelaufzählung der Elemente ist nicht notwendig, weil bereits eine innere abzählbare Ordnung vorliegt. Es genügt daher die Angabe des unteren und oberen begrenzenden Elements. Dazwischen stehen genau zwei Punkte. Negative Integerzahlen sind damit aber ausgeschlossen, denn sie passen nicht in diese (positive) Ordnung.

Es ist aber auch – wie immer in Pascal – möglich, eigene Typen zu definieren und z.B. alle Wochentage in einer Menge zusammenzufassen. Dazu bieten sich vor allem folgende Möglichkeiten an:

Beispiel 2:
```
Tage= Set Of (Montag,Dienstag,Mittwoch,Donnerstag,
Freitag, Samstag, Sonntag);
```

Die einzelnen Elemente werden also zwischen runde Klammern gesetzt und erhalten mit der einmal erfaßten Reihenfolge ihre Ordnung. »Mittwoch« hat in unserem Beispiel somit die Nummer zwei.

Das läßt sich jederzeit überprüfen, indem Sie die Funktion **Ord** darauf anwenden: **WriteLn**(**Ord**(Mittwoch)); schreibt richtigerweise »2« auf den Schirm.

Die zweite Möglichkeit kann so aussehen:

Beispiel 3:

```
TYPE
  TageTyp=(Montag, Dienstag, Mittwoch, Donnerstag, Freitag,
           Samstag, Sonntag);
  TageMenge=Set Of Montag..Freitag;
```

Die bereits vorhandene Ordnung im »TageTyp« wird zur Definition einer Menge verwendet.

Zusammenfassung und Ergänzungen:

- Mengendeklarierungen können verkürzt vorgenommen werden, wenn eine abzählbare Ordnung der Elemente bereits vorliegt, ansonsten müssen die Elemente einzeln erfaßt werden.

- Um mit Mengen arbeiten zu können, benötigen wir entweder Mengenvariablen oder Mengenkonstanten. Sie werden wie üblich deklariert:

  zu Beispiel 1: `VAR menge1,menge2: MengenTyp;`

  Die Variablen »menge1« und »menge2« sind nun in der Lage, beliebige Elemente aus dem Integerbereich von 0 bis 255 aufzunehmen.

  Mit `CONST Grundmenge: MengenTyp=[0..100];`

  wird dagegen der Variablen »Grundmenge« zunächst einmal eine Menge zugewiesen, die in unserem Fall genau 101 Elemente enthält, nämlich die Ganzzahlen von 0 bis 100.

- Beachten Sie bitte die Schreibweisen: Bei der Festlegung der Mengentypen wird bei der aufzählenden Schreibweise mit runden Klammern, bei der Intervallschreibweise dagegen mit zwei Punkten zwischen Ober- und Untergrenze in eckigen Klammern gearbeitet.

- Dagegen werden immer dann, wenn einer Menge irgendwelche Elemente zugewiesen werden, eckige Klammern verwendet. Das entspricht zwar nicht den mathematischen Normen, denn dann müßten bei der aufzählenden Schreibweise die geschweiften Klammern verwendet werden. Turbo dagegen kennt nur die eckigen Intervallklammern, die sowohl das unterste als auch das höchste Element mit einschließen, also ein abgeschlossenes Intervall bilden.

16.3 Operationen mit Mengen Beispiel: Lösung von Ungleichungen

Um Ihnen die Mengenverknüpfungen und -operationen an einem sinnvollen Beispiel nahezubringen, haben wir uns eine Aufgabe aus der Mathematik vorgenommen:

Quadratische Ungleichungen lassen sich (sofern die entsprechende Gleichung überhaupt Lösungen in R hat) in folgende Form bringen:

`(x+a) (x+b) > 0`	beziehungsweise
`(x+a) (x+b) < 0`	oder ähnliches.

Die Parameter a und b sind dabei beliebige reelle Zahlen, die wir einschränkend nur als ganze Zahlen zulassen wollen.

Beispiel: `(x+3) (x-2) >= 0` bedeutet, es sind Belegungen für x gesucht, die diese Ungleichung erfüllen.

Geben wir dazu noch an, daß diese Belegungen aus der Grundmenge [-4..+3] stammen sollen, dann lassen sich durch probierendes Einsetzen die Lösungselemente ermitteln: Die Zahlen -4, -3, 2 und 3 führen jeweils zu einer wahren Aussage.

Wie kann man nun solche Aufgaben mit Turbo lösen?

Dazu führen wir zunächst folgende Überlegungen durch:

- Die Grundmenge, aus der unsere Lösungselemente stammen dürfen, sei eine Menge ganzer Zahlen von einer unteren bis zu einer oberen Grenze. Sie hat die Form [unten..oben].

- Der linke Ausdruck der Ungleichung (x+3)(x-2) ist ein Produkt, der rechte Ausdruck sei immer Null.

- Ein Produkt aus zwei Faktoren ist immer dann größer Null (positiv), wenn

  a) beide Faktoren positiv (größer Null)
  b) beide Faktoren negativ (kleiner Null)

  sind.

- Daraus ergeben sich zwei Bedingungen für die Lösungselemente:

  a) Alle Zahlen für die gilt

     x>=-3 und zugleich x>=2

  b) aber auch alle Zahlen für die gilt

     x<=-3 und zugleich x<=2

- Zu a) erhält man eine erste Lösungsmenge L1, wenn man alle Zahlen aus der Grundmenge aussucht, die größer oder gleich -3 sind, und überprüft, ob sie auch größer oder gleich 2 sind.

  Das Ergebnis dafür liefert die sogenannte Schnittmenge aus [-3..oben] und [2..oben].

- Entsprechend ergibt sich zu b) eine zweite Teillösungsmenge L2 aus [unten..-3] und [unten..2].

- Die Gesamtlösungsmenge L erhält man, wenn man die beiden Teillösungsmengen L1 und L2 vereinigt.

- Wenn das Ungleichheitszeichen nicht »>=«, sondern »<« lautet, dann ergeben sich als Lösungselemente genau diejenigen aus der Grundmenge, die bei der Ungleichung mit »>=« nicht dabei waren. Das ist die sogenannte Komplementärmenge oder das Komplement von L.

  Das bedeutet nichts anderes, als daß wir zum Lösen von Ungleichungen nur ein Verfahren erarbeiten müssen, und je nach Bedarf entweder die unveränderte Lösungsmenge oder aber ihr Komplement ausgeben können.

Soviel zu den theoretischen Vorüberlegungen (ohne die eine vernünftige Programmierarbeit nicht möglich ist). Nun zum praktischen Teil, dem entsprechenden Turbo-Text.

Wie wir inzwischen gelernt haben, fassen wir Sinnabschnitte zu Prozeduren zusammen und gestalten ein einfaches Hauptprogramm:

```
BEGIN
  Eingaben;
  Bearbeitung;
  Ausdruck;
END.
```

Dazu benötigen wir eine Reihe von Mengen- und Integervariablen, die wir vorab deklarieren müssen:

```
TYPE
  MengenTyp=Set Of 0..255;
VAR
  Grundmenge,kompl,L1,L2,L: MengenTyp;
  Art,unten,oben,a,b,i,korrekt: Integer;
```

Wozu diese Variablen verwendet werden, erklären wir bei der Besprechung der Prozeduren.

16.3.1 Die Mengenverknüpfungen

Bevor wir die Anweisungsteile der einzelnen Prozeduren aufbauen können, müssen wir uns erst über die Verwendung der Turbo-Operationen im klaren sein. Die üblichen Verknüpfungen von Mengen werden in Turbo mit folgenden Verknüpfungszeichen vorgenommen. Sie entsprechen dabei nicht unbedingt den mathematischen Symbolen:

- Bilden einer *Schnittmenge* S aus zwei Mengen M1 und M2:

```
S:=M1*M2;
```

- Bilden einer *Vereinigungsmenge* V aus zwei Mengen M1 und M2:

```
V:=M1+M2;
```

- Bilden der *Komplementärmenge* K zu einer Menge M über einer Grundmenge G:

```
K:=G-M;
```

- Prüfung, ob ein Element *in einer Menge M enthalten* ist:

```
IF element IN M THEN Write ('ist Element');
```

- Prüfung, ob eine Menge T *Teilmenge* einer Menge M ist:

```
IF T<=M THEN Write ('ist Teilmenge');
```

- Prüfung, ob eine Menge O *Obermenge* der Menge M ist:

```
IF O>=M THEN Write ('ist Obermenge');
```

- *Gleichheit* oder *Ungleichheit* zweier Mengen werden ganz normal mit »=« bzw. »<>« geprüft.

16.3.2 Der Eingabeteil P71.SET: Korrekturglieder

Der Einfachheit halber schauen wir uns nur zwei der vier möglichen Ungleichungsarten an, nämlich die mit der Relation »>=0« und »<0«. Die beiden anderen mit »<=0« und »>« können Sie bei Bedarf sicher selbst lösen, wenn Sie sich dafür interessieren.

Die Eingabeprozedur hat dann folgendes Aussehen:

```
PROCEDURE Eingaben; (* P71.set *)
BEGIN
  ClrScr;korrekt:=0;l1:=[];l2:=[];
  WriteLn('Loesung von quadrat. Ungleichungen der Form:');
  WriteLn('(x+a)(x+b) >= 0                    <0>');
  WriteLn('(x+a)(x+b) <  0                    <1>');
  Write(^J^M,'Aufgabenart auswaehlen:          ');
  ReadLn(art);
  Write('    mit a: ');
  ReadLn(a);
```

```
  IF a>korrekt THEN korrekt:=a;
  Write('     mit b: ');
  ReadLn(b);
  IF b>korrekt THEN korrekt:=b;
  Write(^J^M,'Grundmenge kleinstes Element: ');
  ReadLn(unten);
  IF Abs(unten) > korrekt THEN korrekt:=Abs(unten);
  Write('Grundmenge groesstes Element: ');
  ReadLn(oben);
  Write(^J^M,'(x+',a,')(x+',b,')');
  IF Art=1 THEN Write(' < ') ELSE Write(' >= ');
  WriteLn('0 hat die Loesungen: ');
END;
```

Erläuterungen:

Sie finden bei den Eingaben immer wieder das Korrekturglied »korrekt«. Wir benützen es dazu, den gesamten Elementebereich ausschließlich im positiven Bereich zu behalten. Denn falls bei den Eingaben irgendwelche negativen Zahlen gewünscht werden sollten, verschieben wir kurzerhand alles so weit, bis alle zu bearbeitenden Elemente positive Ordnungszahlen haben. Unsere Grundmenge ist ja nur von 0 bis 255 definiert. Mehr ist nicht möglich, und der negative Bereich darf bei Integermengen nicht verwendet werden.

Wird zum Beispiel bei der Grundmenge (siehe unter Punkt C) eine negative Untergrenze eingegeben, dann nimmt das Korrekturglied den entsprechenden Betrag an. Später werden dann alle Mengen um diesen Wert nach oben verschoben, so daß nur positive Zahlen auftreten. Beim Ergebnis muß das Ganze dann wieder rückgängig gemacht werden.

(A) Eine kleine Abfrage läßt die Auswahl zwischen zwei Ungleichungen zu. Die Art der Ungleichung wird in der Integervariablen »art« festgehalten.

(B) Nun werden die Werte für die gewünschten Formvariablen a und b erfaßt. Nehmen wir an, daß für a=4 und für b=-2 eingegeben wurde, dann ergibt sich für die erste Art (art=0) die Ungleichung: $(x+4)(x-2) >= 0$

Nachdem das Glied b nun negativ ist, wird das Korrekturglied entsprechend belegt, und zwar so, daß nur dann ein neuer Korrekturwert notwendig wird, wenn ein noch kleinerer Eingabewert als ein bereits korrigierter auftritt.

(C) Bei der Eingabe der Grenzwerte für die Grundmenge ist dabei nur die untere Grenze entscheidend für eine eventuelle Änderung des Korrekturwertes.

Eine Absicherung gegen die Eingabe eines Bereiches, der mehr als 256 Elemente umfaßt, wurde hier nicht vorgenommen. Sie können das selbst erledigen.

(D) Als Abschluß des Eingabeteils drucken wir die erfaßte Ungleichung aus. Hier tritt bei negativen Werten noch die Folge »+-« auf, was dann insgesamt minus ergibt. Auch hier können Sie gegen diesen kleinen Schönheitsfehler eigene Schritte unternehmen, wenn Sie wollen.

Insgesamt wurde zwar auf einen ordentlichen Bildschirmaufbau geachtet, auf großen Eingabekomfort haben wir verzichtet, um das Wesentliche nicht zu überdecken. Schließlich wollen wir hier ja nur zeigen, wie man mit Mengen operieren kann.

16.3.3 Der Lösungsteil: P72.SET

Die Prozedur, die die Ungleichung löst, nennen wir »Bearbeitung«. Zum besseren Verständnis beziehen sich die Erklärungen auf das Beispiel aus 16.3.2 mit den dort genannten Parametern:

```
(x+4) (x-2) >= 0
```

und mit der Grundmenge

```
= [-6..8].
```

Das Korrekturglied »korrekt« muß dann +6 sein, da mit der unteren Grenze der Grundmenge das kleinste mögliche Element festgelegt wurde. Der Ablauf der Lösungsprozedur ist dann folgender:

```
PROCEDURE Bearbeitung; (* P72.SET *)
BEGIN
  unten:=unten+korrekt;
  oben:=oben+korrekt;
  Grundmenge:=[unten..oben];
  a:=-a+korrekt;
  b:=-b+korrekt;
  l2:=[unten..a]*[unten..b];
  l1:=[a..oben]*[b..oben];
  l:=l1+l2;
```

```
    IF art=1 THEN
    BEGIN
      l:=Grundmenge-l;
      l1:=[];
      l2:=l;
    END;
  END;
```

Erläuterungen:

(A) Unter- und Obergrenze der Grundmenge werden zuerst so korrigiert, daß alle Elemente im positiven Bereich liegen. Gerechnet wird intern zunächst mit

```
unten=0;
oben=14;
Grundmenge=[0..14];
```

(B) Die Lösungen der entsprechenden quadratischen Gleichung sind immer (-a) und (-b). Damit werden gleichzeitig auch die Grenzen der Intervalle für unsere Ungleichung festgelegt, die wir nun aber wegen der veränderten Grundmenge ebenfalls mit +6 korrigieren müssen:

```
a=2;   (aus -(+4)+6);
b=8;   (aus -(-2)+6);
```

(C) Eine erste vorläufige Lösungsmenge L1 ergibt sich aus der ersten Schnittmenge

```
L1 = [2..14] * [8..14];
```

Der zweite (vorläufige) Lösungsteil L2 muß alle Elemente berücksichtigen, die kleiner oder gleich groß sind wie die Grenzwerte:

```
L2 = [0..2] * [0..8];
```

Für unser Beispiel ergeben sich daraus

```
L1 = [8..14];
L2 = [0..2];
```

(D) Die Gesamtlösungsmenge L erhalten wir, wenn wir die beiden Teillösungsmengen L1 und L2 vereinigen:

`L=[8..14] + [0..2];` ergibt `L=[0..2,10..14];`

(E) Falls bei der Ungleichung das Relationszeichen »<« gelautet hätte, hat sie genau die Elemente der Grundmenge zur Lösung, die bei dem eben besprochenen Fall nicht aufgetreten sind. Wir erhalten dieses Komplement so:

`L=Grundmenge-L;` also `L=[3..7];`

Übrigens wäre dann eine der beiden Teillösungsmengen leer und die zweite enthielte bereits die Gesamtlösungsmenge. Wir brauchen dies nicht noch einmal durchzurechnen, sondern können diese Teilergebnisse gleich ausgeben. Der Anwender merkt davon nichts.

16.3.4 Der Lösungsausdruck Prozeduren P73.SET und P74.SET

Beim Ausdruck der Lösung dürfen wir aber nicht vergessen, daß wir alle Berechnungen mit dem Korrekturglied »korrekt« durchgeführt haben. Jetzt müssen wir wieder dafür sorgen, daß die Ergebnisse um diesen Wert wieder zurückkorrigiert werden:

```
PROCEDURE Ausdruck; (* P73.SET *)
BEGIN
  Write(^J^M,'L1 = ');Druck(l1);
  Write('L2 = ');Druck(l2);
  WriteLn;
  Write('L  = ');Druck(l);
  WriteLn;
  IF l<=Grundmenge THEN
  WriteLn('Loesungsmenge ist in der Grundmenge enthalten.');
  IF l>=Grundmenge THEN
  Write('Loesungsmenge ist gleich Grundmenge.');
END;
```

Erläuterungen:

(A) Wir geben zur Kontrolle auch die Teillösungen L1 und L2 aus, und zwar in aufzählender Schreibweise, so daß alle Elemente genannt werden, die aus der Ungleichung eine wahre Aussage machen.

Da dieser Ausdruck mehrmals erfolgen muß, erstellen wir dazu eine eigene Prozedur »Druck«, wie sich das in Pascal gehört. Sie überprüft alle Elemente der Grundmenge auf ihr Vorhandensein in der entsprechenden Lösungsmenge und gibt sie mit einem Komma als Trennzeichen aus.

Für den normgerechten Ausdruck von Mengen können wir nun auch die geschweiften Klammern verwenden, die von Turbo nicht als Kommentarklammern erkannt werden, wenn sie innerhalb einer Zeichenkette stehen:

```
PROCEDURE Druck (ld:MengenTyp); (* P74.SET *)
BEGIN
  Write('{ ');
  FOR i:=unten TO oben DO
  IF i IN ld THEN Write(i-korrekt:3,',');
  WriteLn(^H,' }');
END;
```

Mit »i-korrekt« erfolgt dabei die Bereinigung der Korrektur.

(B) Hier soll noch einmal kurz die Verwendung der Operationen mit Teilmengen (<=) und Obermengen (>=) demonstriert werden. Beim letzten Ausdruck könnte man ebensogut schreiben:

```
IF L=Grundmenge THEN Write ...
```

Insgesamt erhalten wir für unser Beispiel als Ergebnisse:

```
L1={2,3,4,5,6,7,8}
L2={-6,-5,-4}
L={-6,-5,-4,2,3,4,5,6,7,8}
```

Auch wenn Mengenlehre nicht Ihr Lieblingsthema ist, sollten Sie dennoch die einzelnen Programmteile zu einem lauffähigen Programm zusammenstellen und überprüfen, ob Turbo auch richtig arbeitet, bzw. ob mit Turbo richtig programmiert wurde.

Zusammenfassung:

- Das Arbeiten mit Mengen erfolgt nach den mathematischen Regeln, jedoch sind bei Turbo Einschränkungen bezüglich der Mächtigkeit der Mengen und der Elemente-Vielfalt hinzunehmen.

Kapitel

17

Verknüpfung von Programmen

In diesem Kapitel werden wir nicht nur den Zusammenbau von selbsterstellten Programmteilen besprechen. Vielmehr stecken wir den Rahmen etwas weiter ab und erklären auch, wie man mit Turbo die bereits im Betriebssystem vorhandenen Routinen für eigene Zwecke anzapfen kann.

Grundsätzlich bieten sich zur Aufnahme von Programmteilen folgende Möglichkeiten an:

- Die Einbindung von Pascal-Prozeduren und -Funktionen mit Hilfe der Include-Anweisung. Diese Form haben wir bereits ausführlich in Kapitel 5 bearbeitet.
- Der Aufruf von Routinen des Betriebssystems mit Hilfe von Turbo-Anweisungen.
- Die Einfügung von Maschinenprogrammteilen in den Turbo-Text mit Hilfe der **Inline**-Anweisung.
- Die Aufnahme von selbstentwickelten Maschinenprogrammen, die außerhalb des Turbo-Programms stehen, sogenannte **Externals**.

- Verkettungen von Turbo-Programmen mit **Chain** und **Execute**.

- Die Unterbringung von mehreren Unterprogrammen in einem gemeinsamen Speicherbereich mit Hilfe der Overlay-Technik.

- Kombinationen aus den genannten Punkten.

In den folgenden Abschnitten müssen wir uns auf das Wesentliche beschränken, weil es nicht Sinn dieses Buches sein kann, ausführlich auf die Eigenschaften des Betriebssystems einzugehen oder gar einen Kurs zur Programmierung in Maschinensprache bzw. in Assembler aufzubauen.

Wer hier richtig einsteigen will, der muß sich für sein Gerät auf jeden Fall mit dem Technischen Handbuch, der Programmierung des Prozessors, und vor allem mit der Dokumentation des Betriebssystems befassen.

Wir werden aber versuchen, auch denjenigen eine kleine Hilfe oder wenigstens ein paar Anregungen zu geben, für die der Umgang mit Maschinenbefehlen oder mit den Systemfunktionen noch fremd ist.

Die folgenden Programmbeispiele sind deshalb durchweg praktischer Natur, d.h., wir haben diejenigen ausgewählt, die Sie später bei Ihrer eigenen Programmierarbeit auch wirklich verwenden können. Um es gleich vorwegzunehmen: Turbo ist eigentlich so schnell, daß Sie in den allermeisten Fällen auf den Einsatz von Maschinenspracheteilen verzichten können. Lediglich an einige spezielle Punkte kommt Turbo nicht heran, wenn es nicht zusätzlich durch direkte Befehle an den Prozessor unterstützt wird. Und genau solche Fälle schauen wir uns an.

17.1 Einige Hinweise zum Betriebssystem

Wenn ein Turbo-Programm gestartet wird, dann spielt sich die Verarbeitung der Befehle in mehreren Ebenen ab:

- Die niedrigste Ebene ist die Prozessorebene, wo in elementaren Schritten die einzelnen Bits der Speicherstellen über die Adreßleitungen angesprochen werden.

- Auf der Ebene des Betriebssystems des Rechners werden jeweils eine Reihe von Maschinenbefehlen zu Routinen zusammengefaßt, mit denen man komplexe Anweisungen an den Prozessor übergeben kann. Das gilt auch für die höheren Programmiersprachen wie z.B. BASIC.

Das Betriebssystem CP/M verwendet über das sogenannte BIOS, das BASIC Input Output System, u.a. auch die in den ROMs des Rechners gespeicherten Routinen, während der zweite Teil von CP/M, das BASIC Disk Operating System, kurz BDOS, eine eigenständige Dateiverwaltung betreibt, die wiederum mit dem BIOS in Kontakt steht.

Die dritte Ebene ist nun die der Programmiersprache, in unserem Fall Turbo-Pascal. Und Turbo bedient sich über CP/M der Routinen des Betriebssystems und kann auch bis in die Prozessorebene »hinuntergreifen«.

Das mag für unsere Zwecke genügen. Für einen tieferen Einstieg in diese Materie sollten Sie sich ein Buch über CP/M Plus zu Gemüte führen.

17.2 Der Prozessor des CPC 6128

Da wir in den nächsten Abschnitten auch einige Teile in Assembler vorstellen werden, sollten Sie wenigstens ganz grob über das Herz Ihres Computers informiert sein. Es ist die CPU (Central Processing Unit) Z80A von Zilog.

Alle Register sind 8-Bit-Register, wobei aber jeweils zwei zusammenhängende gemeinsam angesprochen werden können, so daß mit etlichen Maschinenbefehlen nicht nur ein Byte (8 Bits), sondern ein Wort (2 Byte oder 16 Bit) übertragen werden kann.

Bei den Assemblerbeispielen sollten Sie das Registerschema vor Augen haben:

A	F	A'	F'	AF- bzw. A'F'-Register
B	C	B'	C'	BC- bzw. B'C'-Register
D	E	D'	E'	DE- bzw. D'E'-Register
H	L	H'	L'	HL- bzw. H'L'-Register
I	R	Interrupt Vektor und Memory Refresh		
IX		Index-Register IX		
IY		Index-Register IY		
SP		Stackpointer		
PC		Programmzähler (Program Counter)		

Bild 17-1: Registerschema des Z80A

Wenn Sie sich mit Assembler beschäftigt haben, wird Ihnen das nichts Neues sein, ansonsten sollten Sie einmal in einem Programmierhandbuch für Ihren Prozessor blättern.

Uns genügt vorläufig dieses Grundwissen, um uns mit den Systemaufrufen und Maschinenprogrammen zu beschäftigen.

17.3 Die Speicherverwaltung

Da durch einen 8-Bit-Prozessor wie den Z80 immer nur ein Speicherbereich von höchstens 64 Kbyte (2 hoch 16, also 65536 Adressen) mit Hilfe eines Lo- und eines Hi-Bytes angesprochen werden kann, werden die 128 Kbyte in 8 Blöcken zu je 16 Kbyte verwaltet.

Würde man die 128 Kbyte des CPC 6128 durchzählen, ergäbe sich folgende Aufteilung:

Block 0: von $0000 bis $3FFF
Block 1: von $4000 bis $7FFF
Block 2: von $8000 bis $BFFF
Block 3: von $C000 bis $FFFF
Block 4: von $10000 bis $13FFF
Block 5: von $14000 bis $17FFF
Block 6: von $18000 bis $1BFFF
Block 7: von $1C000 bis $1FFFF

Der Prozessor ist aber nur in der Lage, jeweils vier 16-Kbyte-Blöcke zu überblicken, die er stur mit seinen Adreßzählern von 0 bis 65535 verwaltet. Welche Blöcke er dabei erwischt, ist ihm egal.

Damit hier Ordnung herrscht, muß das Betriebssystem bzw. der Programmierer dafür sorgen, daß die richtigen Blöcke zusammengestellt werden. Immer vier Blöcke ergeben eine Bank.

Beim CPC 6128 sind 8 solcher Bänke vorgesehen, die von 0 bis 7 durchgezählt werden und folgende Zusammenstellungen aufweisen:

Bank	0	1	2	3	4	5	6	7
mit Blocknummern	3	7	7	7	3	3	3	3
	2	2	6	2	2	2	2	2
	1	1	5	–	4	5	6	7
	0	0	4	0	0	0	0	0

Bild 17-2: *Speicherverwaltung des CPC 6128*

Welche Bank gerade aktiv ist, wird in der Systemadresse $B8D5 vermerkt. Umgeschaltet wird die Bank über das sogenannte Gate Array, das über die Adresse $7FXX beschrieben wird. XX ist dabei das Lo-Byte der Adresse. Es hat den Wert $C0 (=192), wenn Bank 0 aktiviert werden soll. $C1 steht für Bank1, $C2 für Bank 2 usw.

Gesendet wird die Bank mit dem Assemblerbefehl OUT.

Für unsere Arbeit mit Turbo-Programmen ist eigentlich nur folgendes wichtig:

CP/M verwendet normalerweise die Bank 1 mit den Blöcken 0,1,2 und 7, bedient sich dabei aber auch der CPC-Routinen aus dem Block 3 in der Bank 0. Das bedeutet, daß unter CP/M auch ein sogenanntes Bankswitching stattfindet, um zwischen den beiden Konfigurationen wechseln zu können, was aber nicht allzu schwierig ist, da Bank 0 und Bank 1 mit den Blöcken 0, 1 und 2 eine sehr große Schnittstelle (common area = gemeinsamer Bereich) haben.

Turbo dagegen arbeitet von Bank 2 aus mit den Blöcken 4, 5, 6 und 7. Wie man sieht, kann Turbo nun über die gemeinsame Bank 7 mit CP/M kommunizieren, zumal CP/M in dieser Bank 7 nur die Adressen ab $F500 bis $FFFF belegt hat.

Also gelingt es mit Turbo, auch auf die Blöcke 0, 1, 2 und 3 zuzugreifen, was sehr wichtig ist, weil z.B. der Bildschirmspeicher unter CP/M nicht mehr an der Stelle zu finden ist, wo er z.B. von BASIC aus gesehen liegt. Wir kommen demnächst auf dieses Thema zu sprechen. An dieser Stelle sei nur gesagt, daß sich der Bildschirminhalt im zweiten Block (Block 1), also unter Bank 1 ab der Adresse $4000 finden läßt.

17.4 Untersuchung von Routinen des Betriebssystems

Wenn man sich die Routinen eines Systems wie Turbo oder CP/M genauer anschauen will, dann tut man das in der Regel mit einem sogenannten Monitor oder einem Disassembler, der den Maschinencode in die leichter verständlichen Assembler-Befehle (Mnemonics) übersetzt. Leider gelingt es nur auf Umwegen, auch dann in die von Turbo benützten Speicherblöcke 4 bis 7 zu schauen, weil die meisten Disassembler in der Bank 0 installiert sind.

Für die Erforschung kleinerer Teilstücke bietet sich aber folgendes Verfahren an, dessen wir uns auch bedient haben, um die eine oder andere Einsprungstelle ausfindig zu machen:

1. Schritt:
Mit Hilfe eines Turbo-Programms wird ein Adreßbereich ausgelesen und auf dem Drucker in Form eines Hex-Dumps (Auflistung der Speicherinhalte in Hexzahlen) ausgegeben.

2. Schritt:
Mit einem Disassemblerprogramm, wie z.B. MONA, lassen sich die ausgedruckten Zahlen in einen freien Speicherbereich übertragen und in Form eines Assemblerlistings anschauen.

Alternative:
Der Hex-Dump wird auf Diskette geschrieben und ein geeignetes Disassemblerprogramm holt sich von dort wieder die Daten und ermöglicht ein Assemblerlisting. Damit spart man sich das Eintippen des Hexcodes. Das Problem ist nur, daß die von Turbo erzeugte Datei nicht immer kompatibel (verträglich) mit der vom Assembler generierten Monitordatei ist. Hier hilft eigentlich nur die eigene Untersuchung, ob der Assembler zu Turbo paßt.

Bleiben wir bei der immer möglichen ersten Form und lassen uns jedenfalls Turbos Eingeweide zeigen. Das folgende Programm kann das. Wir greifen dabei auf Turbo-Operationen zurück, die wir erst erklären müssen. Es handelt sich dabei um den Gebrauch von Operatoren, die mit Bits manipulieren. Wir kennen davon schon **AND**, **OR** und **NOT**.

Außerdem kann man einen Speicherinhalt nach links oder rechts bitweise versetzen (shiften). Die Operatoren dazu lauten **SHL** bzw. **SHR**.

Dazu ein Beispiel:

Ein Byte hat den Wert 65. Dann sieht die Bitfolge, beginnend mit dem höchstwertigen, wie folgt aus: 01000001.

Der Befehl 65 **SHR** 4 bewirkt, daß alle Bits um vier Stellen nach rechts versetzt werden, so daß sich folgende Stellung ergibt: 00000100, was einem Wert von 8 entspricht. Die links freiwerdenden Stellen werden mit Nullen aufgefüllt.

Zusammengefaßt ergibt sich: `65 SHR 4 = 8`

SHL funktioniert entsprechend: `65 SHL 4 = 16`, weil 01000001 **SHR** 4 = 00010000.

Daneben existiert auch noch das exklusive Oder, genannt **XOR**, wo bei der Verknüpfung von Bits als Ergebnis immer nur dann ein Bit gesetzt wird, wenn entweder der eine oder der andere Operator ein gesetztes Bit liefert.

Beispiel: `65 XOR 35 = 98`, weil 01000001 **XOR** 00100011 = 01100010

Das mag als Abschluß zum Thema Bitverknüpfungen genügen.

Schauen wir uns nun ein Programm an, das aus einem bestimmten Speicherbereich den Maschinencode in Form von Hexzahlen darstellt.

```
       PROGRAM HexDump; (* P76.PAS *)
       TYPE
(*A*)    L4=String[4];
       VAR
(*B*)    i,k,l,m:Integer;

(*C*) FUNCTION HexZahl(dezzahl:Integer):l4;
       CONST
(*D*)    hexziffer:Array [0..15] Of Char ='0123456789ABCDEF';
       VAR
(*E*)    teil:Byte;
         mani:String[2];
      BEGIN
(*F*) teil:=Hi(dezzahl);
(*H*) mani:=hexziffer[teil SHR 4] + hexziffer[teil AND $0F];
(*I*) teil:=Lo(dezzahl);
(*J*) HexZahl:={mani+}hexziffer[teil SHR 4]+hexziffer[teil AND $0F];
      END;

       BEGIN
(*K*)    Write('ANFANG: ');ReadLn(l);
         Write('ENDE:   ');ReadLn(m);
         k:=-1;
          FOR I:=l TO m DO
         BEGIN
(*L*)      k:=k+1;IF k=20 THEN BEGIN k:=0;WriteLn(Lst);END;
           Write(HexZahl(Mem[I]),' ');
(*M*)      Write(Lst,HexZahl(Mem[I]),' ');
         END;
         WriteLn(Lst,^J^J^J^M^Z);
       END.
```

Erläuterungen:

(A) Für eine Hexzahl im Adreßbereich bis $FFFF benötigen wir Stringtypen mit vier Stellen. Auf einen Vorsatz wie $ oder Nachsatz wie H verzichten wir bei der Ausgabe.

(B) Für die Zählschleifen benötigen wir die üblichen Integervariablen.

(C) Die Funktion »HexZahl« verwandelt einen Bytewert, den wir als »dezzahl« übergeben, in einen Hex-String. Wenn z.B. 32768 übernommen wurde, soll als Funktionsergebnis 8000 ausgegeben werden.

(D) Dazu stellen wir in einem **Array** »hexziffer« die möglichen Ziffern der Hexzahlen wohlgeordnet bereit. Beachten Sie die hier mögliche Schreibweise des Zeichen-**Array**s.

(E) Die umzuwandelnde Integerzahl bearbeiten wir in zwei Schritten, weil sie im Speicher stets als Zweibyte-Zahl verwaltet wird. Dazu werden die Variablen »teil« und »mani« gebraucht.

(F) Die Standardfunktion **Hi** liefert das Hi-Byte einer Integerzahl als neue Integerzahl. Aus 32768 (=$8000) wird z.B. $0080 = 128.

(G) Jetzt erfolgt die Manipulation mit den Bits: Der Hexstring für unser Hi-Byte besteht aus zwei Hex-Ziffern. Shiften wir den Byte-Inhalt um vier Stellen (weil mit vier Bit jede Hexziffer erfaßt werden kann), dann läßt sich aus der **Array**-Konstanten genau die Ziffer mit dem entsprechenden Index entnehmen.

Die zweite Stelle erhalten wir, indem wir aus dem Byte die ersten vier Bit ausschalten mit **AND** 00001111 (= **AND** $0F).

Zusammengefaßt: Wir betrachten einen Bytewert als zweistellige Hexzahl, wobei die erste Stelle die vier oberen Bits, die zweite Stelle die vier unteren darstellt.

(H) Das gleiche geschieht nun mit dem Lo-Byte der umzuwandelnden Zahl, wobei die Standardfunktion **Lo** die Zerlegung übernimmt, indem sie von einer Integerzahl nur den Lo-Wert übernimmt.

(I) Die Zusammensetzung zur kompletten Hexzahl wird gleich mit vorgenommen, indem wir den »Hi-String« mit dem »Lo-String« verknüpfen. Wenn nur Bytes

umgewandelt werden sollen, läßt man den Hi-Teil eben weg, weil sonst immer zwei Nullen vorangestellt würden.

(J) Der Hauptteil dient zunächst der Abfrage der Anfangs- und der Endadresse.

(K) Nach jeweils 20 Hexzahlen erfolgt ein Zeilensprung. Ansonsten wird mit **Mem** die gewünschte Speicherstelle untersucht.

(L) Wenn Sie die Ausgabe nicht auf dem Drucker wünschen, dann entfernen Sie **LST** aus den **Write**-Anweisungen oder leiten Sie die Ausgabe auf eine Diskettendatei um.

Dieses Programm ist für Wissensdurstige recht wertvoll, so daß es sich lohnen könnte, dazu ein kleines Menü zu erstellen, mit dem man die Art der Ausgabe abrufen kann: Schirm, Drucker oder Diskette. Wer den Maschinencode des Z80 beherrscht, kann sich auf diese Weise auch gleich ein Disassembler-Programm mit Turbo schreiben.

17.5 Integrierte Maschinenprogramme Prozedur P77.GRA: Inline, Betriebssystemroutinen Programm P78.DEM: Grafik-Demo

Turbo erlaubt es, in den Pascal-Text hinein Maschinenprogramme zu schreiben, die beim Kompilieren gleich mit integriert werden und ablauffähig sind.

Dazu wird in einer **Inline**-Anweisung der Maschinencode in Form von Zahlen und Variablen verarbeitet. Leider steht dafür kein Assemblerprogramm zur Verfügung, so daß nicht mit den komfortablen Assembler-Kürzeln (den sogenannten Mnemonics) gearbeitet werden kann. Die Maschinenbefehle werden im endgültigen Maschinencode erwartet, so wie ihn der Prozessor verarbeiten kann: in Bytes oder in Wörtern.

Die Eingabe der Code-Elemente dagegen kann in beliebiger Form erfolgen:

- mit Dezimalzahlen

  Beispiel: **Inline** (42/20000);
  Bedeutung: LD HL,(20000)
  (Lade das Registerpaar HL mit dem Inhalt aus 20000/20001)

- mit Hexzahlen

  Beispiel: **Inline** ($2A/$204E);
  Bedeutung: (gleiche Bedeutung wie oben)

- gemischt

  Beispiel: **Inline** ($2A/20000);

- mit Hilfe von Variablen

  Beispiel: x:=20000;
  Inline ($2A/x);

Im letzten Fall wird für »x« die Anfangsadresse der Variablen »x« als ein Wort (2 Byte Lo/Hi) in den Maschinencode aufgenommen. Man braucht sich also nicht selbst darum zu kümmern, wo die Variableninhalte im Speicher abgelegt sind.

Zur Form:

- Auch die Form **Inline** ($2A/x+3) ist möglich. Entsprechend wird hier nun auf eine um 3 höhere Adresse als die Anfangsadresse von »x« verwiesen. Die beiden Datenelemente »x« und »3« bilden zusammen mit dem Trennzeichen »+« ein einziges Code-Element.

- Alle Code-Elemente einer **Inline**-Anweisung werden in runden Klammern erfaßt und durch Schrägstriche getrennt.

- Um die Codierung der Maschinenbefehle kümmert man sich zweckmäßigerweise mit Hilfe eines Assemblers, läßt sich ein Listing ausdrucken und tippt dann den Maschinencode in den Turbo-Text ein.

  Das genügt normalerweise, denn mit **Inline** wird man in der Regel nur kurze Maschinenteile übernehmen. Wir werden aber weiter unten die Codezahlen aufführen, die immer wieder benötigt werden und mit denen man **Inline**-Programme aufbauen kann.

Doch nun zu unserem Beispiel-Programm. Wir haben zwar für dieses Buch vereinbart, daß wir Grafikprobleme ausklammern. Wir wollen aber dennoch andeuten, wie Turbo mit Grafik umgeht.

Beispielhaft für alle ähnlichen Fälle greifen wir nun folgendes Problem auf: Mit Hilfe der bereits vorhandenen Grafikroutinen des Betriebssystems soll ein PLOT-Befehl mit Turbo erstellt werden, der an eine bestimmte Bildschirmstelle einen Punkt setzt. Das ist also ein grundlegender Befehl, auf dem sich viele andere Grafikbefehle zum Ziehen von Linien oder ähnlichem aufbauen lassen.

Auf Grund der Tatsache, daß die Betriebsroutinen aber in einer anderen Bank bzw. in ROMs liegen, die erst aktiviert werden müssen, verzweifelt man sicher, wenn man versucht, mit einem **Inline**-Programm die notwendigen Anweisungen zu erteilen. Es müssen vor dem Umschalten die Rückkehradressen gesichert werden, die Stackinhalte brauchen ständige Überwachung usw., so daß ein dickes Maschinencode-Paket entstehen würde. Und beachtet man irgendeine Kleinigkeit nicht, dann findet meistens das **Inline**-Programm nicht mehr zurück, und der Rechner hängt sich irgendwo auf.

Das kann aber nicht der Sinn von **Inline**-Programmen sein. Doch hier gibt es einen eleganten Ausweg: Turbo selbst bedient sich doch über CP/M der Betriebsroutinen und kennt die Schleichwege hin bis zur Ausführung des gewünschten Befehls und zurück zum eigenen Turbo-Programm.

Es gibt in Bank 7 eine Ansprungstelle, die die gewünschte Routine aufruft, wenn man die Einsprungadresse angibt. Stacküberwachung, Bankswitching, Rückkehradressen und alle anderen benötigten Aufrufe von Hilfsroutinen werden automatisch erledigt. Das einzige was man selbst noch tun muß: die für die Betriebsroutine benötigten Parameter bereitstellen. Der Wichtigkeit des Themas angemessen, beschreiben wir das folgende Musterbeispiel ausführlich, so daß Sie in der Lage sind, es für jeden gewünschten Zweck umzubauen.

```
(*A*) PROCEDURE Plotxy(x,y:Integer); (* P77.GRA *)
      BEGIN
(*B*)   InLine($2a/x/        {LD HL,(x)}
(*C*)          $eb/          {EX DE,HL}
(*D*)          $2a/y/        {LD HL,(y)}
(*E*)          $cd/$5a/$fc/  {CALL $FC5A}
(*F*)          $ea/$bb);     {Einsprungadresse $BBEA}
      END;
```

Erläuterungen:

(A) Für die Ausführung der Plot-Anweisung werden die x-Koordinate »x« und die y-Koordinate »y« aus der aufrufenden Routine als Festwertparameter übernommen. Beides sind Integerzahlen.

(B) Der erste **Inline**-Befehl lädt das Registerpaar HL mit der Integerzahl, die in der Variablen »x« abgelegt wurde.

(C) Die x-Koordinate wird aber von der Betriebsroutine im Registerpaar DE erwartet. Also tauschen wir mit EX DE,HL die Inhalte der beiden Registerpaare aus. Das gleiche Ergebnis erzielt man, wenn man gleich DE mit dem etwas längeren Code $ED/$5B lädt. Dafür entfällt aber dann der Vertauschbefehl $EB.

(D) Da die y-Koordinate vom aufzurufenden Programm in HL verlangt wird, holen wir sie dorthin.

(E) Nun rufen wir ein Unterprogramm auf, das ab Adresse $FC5A beginnt. Dort erfolgt aber zunächst wieder nur ein Sprung an eine andere Adresse ($FD10), wo eine Vorbereitungsroutine abgelegt ist, die sich unter anderem auch die endgültig anzulaufende Adresse $BBEA aus dem Turbo-Programm holt und die abschließend dafür sorgt, daß Turbo auch genau hinter der letzten **Inline**-Anweisung weitermacht.

(F) Die Adresse der aufzurufenden Routine wird in der Form Lo/Hi als letzte Information in die **Inline**-Anweisung aufgenommen. Übrigens ist an der Stelle $BBEA auch noch nicht die Routine zu finden, an der der Plot-Befehl »Plot absolute« beginnt. Vielmehr gehört der Bereich von $BB00 bis $BD5D zu der Haupttabelle der sogenannten JUMP-RESTORE-Vektoren; von hier aus erfolgt der Sprung zur gewünschten Routine des Betriebssystems.

Wer sich durch diese Erklärungen überfordert fühlt, braucht sich eigentlich nur folgendes zu merken:

- Über die Adresse $FC5A kann jede Routine des Betriebssystems aufgerufen werden, die in den Tabellen der RAM-Vektoren enthalten ist.

- Die Adresse der gewünschten Routine wird unmittelbar hinter den Code für den Aufruf CALL $FC5A geschrieben (Lo-/Hi-Form!).

- Die Register müssen mit den notwendigen Parametern belegt werden, die man als Übernahmeparameter deklarieren kann.

Woher weiß man aber nun, welche Adresse welche Routine mit welcher Vorbereitung aufrufen kann?

Dazu sollten Sie ein kommentiertes ROM-Listing Ihres CPC 6128 aufschlagen (siehe Literaturangaben). Wir können Ihnen an dieser Stelle nur die wichtigen oder interessanten Einsprünge herausschreiben. Es lohnt sich sowieso nur, diejenigen über ein eigenes Maschinenprogramm aufzurufen, zu denen es keine entsprechenden Turbo-Befehle gibt. Es wäre also z.B. sinnlos, eine Additionsroutine in einem **Inline**-Programm zu entwickeln. Das kann Turbo selbst auch. Was Turbo nicht von Haus aus implementiert hat, sind Grafik- und Sound-Befehle. Und da wollen wir ihm auf die Sprünge helfen:

Name der Routine	Sprungvektor	Vorbereitungen
GRA INITIALIZE	$BBBA	---
GRA RESET	$BBCD	---
GRA MOVE ABSOLUTE	$BBC0	x-Koord. in DE, y-Koord. in HL
GRA MOVE RELATIVE	$BBC3	dto.
GRA WIN WIDTH	$BBCF	links DE, rechts HL
GRA WIN HEIGHT	$BBD2	oben in DE, unten in HL
GRA SET PEN	$BBDE	Farbstift-Nr. in A
GRA SET PAPER	$BBE4	dto.
GRA PLOT ABSOLUTE	$BBEA	x-Koord. in DE, y-Koord. in HL
GRA PLOT RELATIVE	$BBED	dto.
GRA LINE ABSOLUTE	$BBF6	dto.
GRA LINE RELATIVE	$BBF9	dto.
GRA WRITE CHAR	$BBFC	Zeichen in A
GRA FILL	$BD52	A: Farbstift, HL: Pufferadresse, DE: Pufferlänge
GRA SET ORIGIN	$BBC9	DE: x-Koord.; HL: y-Koord.
SCR SET INK	$BC32	A: Farbstift; B: 1.Farbnummer; C: 2.Farbnummer
SCR FILL BOX	$BC44	A: Füllbyte, D: rechts; E: unten; H: links; L: oben;
SOUND RESET	$BCA7	---
SOUND QUEUE	$BCAA	(HL) bis (HL+8): Übergabeblock
SOUND CHECK	$BCAD	A: Kanalmaske
SOUND ARM EVENT	$BCB0	A: Kanalbits; HL: EVENT Block-Adresse
SOUND RELEASE	$BCB3	A: Kanalmaske
SOUND HOLD	$BCB6	---
SOUND CONTINUE	$BCB9	---
SOUND AMPL ENVEL.	$BCBC	A:Hüllkurven-Nr. (HL): ENVELOPE-Folge (16 Byte)
SOUND TONE ENVEL.	$BCBF	dto.

Folgende **Maschinenbefehle** werden Ihnen immer wieder begegnen:

$2A : Lade HL mit dem Inhalt der angegebenen Speicherzelle.
$EB : Vertausche die Inhalte von DE und HL.
$ED5B: Lade DE mit dem Inhalt der angegebenen Speicherzelle.
$3A : Lade A mit dem Inhalt der angegebenen Speicherzelle.
$47 : Lade B mit dem Inhalt von A.
$4F : Lade C mit dem Inhalt von A.
$57 : Lade D mit dem Inhalt von A.
$5F : Lade E mit dem Inhalt von A.

Das mag vorläufig zum Experimentieren genügen. Wenn Sie hier mehr aus Ihren Betriebsroutinen machen wollen, kommen Sie um eine Lektüre zur Programmierung des Z80 nicht herum.

Nun noch ein kleines Demo-Programm zum Ausprobieren unserer **Inline**-Fertigkeiten. Es benützt die oben genannte Prozedur »Plotxy« und ruft sie in einer Schleife mehrfach auf.

Wie Sie im folgenden sehen, lassen sich **Inline**-Programme durchaus auch innerhalb Include-Prozeduren realisieren.

```
      PROGRAM PlotDemo;   (* P78.DEM *)
      VAR x,y,i,j:Integer;

(*A*) PROCEDURE grafik;
      BEGIN
        Write(#27,'0',#27,'y');
      END;

(*B*) PROCEDURE text;
      BEGIN
        Write(#27,'1',#27,'x');
      END;

(*C*) {$I P77.GRA}

      BEGIN
        ClrScr;
(*D*)   Write('X-Anfang (Vorschlag 320): ');
        ReadLn(x);
        Write('Y-Anfang (Vorschlag 100): ');
        Read(y);
        grafik;
```

```
(*E*)    FOR j:=-20 TO 20 DO
         BEGIN
           FOR i:=0 TO 250 DO
           BEGIN
             Plotxy(x+i,y-j*i);
             Plotxy(x-i,y-j*i);
           END;
         END;
         REPEAT UNTIL KeyPressed;
(*F*)    text;
       END.
```

Erläuterungen:

(A) Die Prozedur »grafik« schaltet auf den Grafikschirm um, so daß der Ursprung in der linken unteren Ecke mit den x,y-Koordinaten 0/0 liegt.

(B) »text« schaltet wieder zurück auf den Textschirm und den Rechteck-Cursor.

(C) Die »Plotxy«-Prozedur haben wir auf Diskette ausgelagert.

(D) Wir lassen uns abfragen, mit welchen x,y-Koordinaten begonnen werden soll.

(E) In einer Schleife verändern wir die Parameter, mit denen die »Plotxy«-Prozedur angelaufen wird. Ahnen Sie schon, welches Bild sich auf dem Schirm ergeben wird?

(F) Nach Beendigung der Plotterei kehren wir in den Textschirm zurück, der dadurch auch gelöscht wird.

17.6 Aufruf von CP/M-Routinen Programm P79.DEM: BDOS, Mem, Diskettenspeicherplatz

Das Basic Disc Operating System BDOS arbeitet mit einer ganzen Reihe von Funktionen, die von 0 bis 152 numeriert sind. Jedoch sind davon nur 69 belegt. Die Aufrufe erfolgen über die Adresse 05, in welcher die Sprunganweisung abgelegt ist. Je nachdem, mit welcher Belegung des C-Registers dieser Aufruf erfolgt, wird eine BDOS-Routine abgespult. Steht z.B. 46 in C, dann wird eben Funktion Nummer 46 aufgerufen.

Für unsere Arbeit mit Turbo ist es nur wichtig zu wissen, daß man sich der bereits durch CP/M vorhandenen Maschinenprogramme bedienen kann. Die Anweisung dazu lautet **BDOS**.

Es wäre wiederum Unsinn, wenn man nun versuchte, alle BDOS-Routinen mit Turbo zu bearbeiten. Denn die Turbo-Anweisungen bedienen sich teilweise des BDOS, um z.B. Diskettendateien zu bearbeiten.

Aber auch hier gibt es einige Stellen, an die man mit den einfachen Turbo-Befehlen nicht herankommt. Zum Beispiel gibt es keine Standard-Prozedur, die den noch freien Speicherplatz auf dem eben benützten Laufwerk ausgibt. Erstellen wir uns dazu ein kleines Demonstrationsprogramm, das man jederzeit als eine durchaus nützliche Prozedur in jedes beliebige Turbo-Programm einbauen kann.

Dazu benötigen wir die BDOS-Funktionen 46 und 26.

Um sinnvoll mit diesen Aufrufen arbeiten zu können, müssen Sie auf jeden Fall eine Dokumentation über CP/M zum Nachschlagen haben. Dort finden Sie unter Nummer 46, daß die Funktion den Namen GET DISK FREE SPACE hat, daß sie zum Aufruf im Register E die Nummer des Laufwerks verlangt, und daß der Aufruf dann erfolgt, wenn die Adresse 5 mit einer Belegung von 46 angelaufen wird.

Turbo unterstützt dieses Verlangen, weil es automatisch die Funktionsnummer in das C–Register einsetzt und außerdem eventuell angegebene Parameter in das Registerpaar DE überträgt.

Beispiel: **BDOS**(46,0)
Bedeutung: Rufe BDOS-Routine Nummer 46 mit DE=0/0

Jetzt müssen wir nur noch an das Ergebnis gelangen, das uns diese Funktion liefert. Auch hier gibt eine gute Dokumentation Auskunft:

Es befindet sich in den ersten 3 Adressen des sogenannten DMA-Puffers (Direct Memory Address), dessen Anfangsadresse wir willkürlich wählen können. Und zwar wiederum mit einer BDOS-Funktion, nämlich der Nummer 26.

Zweckmäßigerweise suchen wir dazu einen Speicherbereich aus, der weder von Turbo noch von unserem Programm belegt wird. Oder aber wir greifen auf den standardisierten CP/M-Puffer zurück, der bei $0080 in Block 0 beginnt.

Beispiel: **BDOS**(26,$C100);
Bedeutung: Setze den Anfang des DMA auf Adresse $C100.

Sollten wir die DMA-Adresse wieder auf Standard-Einstellung bringen müssen, dann rufen wir BDOS-Funktion Nummer 13, die einen Disk-Reset durchführt.

BDOS läßt sich von Turbo als Prozedur oder Funktion behandeln. Wird es als Funktion aufgerufen, dann ist das Funktionsergebnis immer der Inhalt des A-Registers nach Durchlaufen der BDOS-Funktion.

Einige BDOS-Funktionen liefern als Ergebnis einen Wert in das Register HL zurück. Auch dafür hat Turbo einen Aufruf. Es ist dies die Funktion **BDOSHL**, die ansonsten die gleiche Wirkung hat wie **BDOS**.

Wie man **BDOS** einsetzt, sehen Sie am Programm P79.DEM:

```
      PROGRAM BDOS_Demo_DiskSpeicherPlatz; (* P79.DEM *)
      VAR
(*A*)   fb:Real;
        drive:Char;
      BEGIN
        BDOS(13);
(*B*)   BDOS(14,0);
(*C*)   IF BDOS(25)=1 THEN drive:='B' ELSE drive:='A';
        Write('auf DRIVE ',drive);
(*D*)   BDOS(26,$C100);
(*E*)   BDOS(46,0);
(*F*)   fb:=(Mem[$C102]*65536.0+Mem[$C101]*256+Mem[$C100])*128;
        Write(' freier Speicherplatz : ',fb:6:0,' Bytes');
      END.
```

Erläuterungen:

(A) Mit der Variablen »fb« geben wir die Anzahl der freien Bytes auf der Diskette an. »drive« nimmt die Laufwerkbezeichnung auf.

(B) Mit der Nummer 14 für **BDOS** läßt sich das Laufwerk bestimmen, und zwar mit dem dahinter angegebenen Parameter. Das heißt, Sie können Ihr Laufwerk auch als Laufwerk B bezeichnen und somit zwei oder auch mehr Laufwerke simulieren, wie Ihnen das die Kopierprogramme von CP/M Plus schon vorgeführt haben.

(C) Hier läuft **BDOS**(25) als Funktion und läßt sich unter diesem Namen auch gleich als Funktionsergebnis weiterverarbeiten. Es gibt an, welches Laufwerk gerade als aktives in Betrieb ist.

(D) Mit **BDOS**(26,$C100) legen wir die Anfangsadresse des DMA auf $C100, einen freien Bereich über unserem Turbo-Text, und wissen damit gleich, wo später die Zahl für den freien Speicherplatz zu holen ist.

(E) Das ist die oben besprochene Prozedur, die nun vom Laufwerk A (oder O) den freien Speicherplatz holen soll.

(F) Das Ergebnis wird in 3 Byte abgelegt, mit drei Stellenwerten im 256er-System. Das bedeutet, daß die höchstwertige Stelle mit 65536, die mittlere mit 256 und die niedrige mit 1 multipliziert werden muß. Zählen wir alles zusammen, ergibt sich der freie Platz als Zahl von Blöcken zu je 128 Byte, so daß wir diese Summe mit 128 multiplizieren und so zum endgültigen Ergebnis kommen.

Aber Achtung: Wenn Sie dieses Programm laufen lassen, werden Sie erstaunt feststellen, daß die Diskette überhaupt nicht anläuft. Das kommt daher, weil unter CP/M eine Directory-Tafel mitgeführt wird, in der alle notwendigen Informationen des aktiven Laufwerks stehen, u.a. auch der freie Speicherplatz. Das führt aber dazu, daß Sie unmittelbar nach dem Einlegen einer neuen Diskette noch die Daten der vorhergehenden im Speicher haben, solange keine Diskettenoperation erfolgt ist.

Sie können dies ändern, indem Sie einen DISK RESET mit **BDOS**(13) als erste Anweisung bei (B) einfügen und haben ab sofort ein Programm, das Ihnen zuverlässig den freien Speicher der tatsächlich im Laufwerk arretierten Diskette anzeigt.

Aus der Vielzahl der BDOS-Funktionen hier eine Auswahl, die für Turbo geeignet sind:

Nr.	Name im BDOS	Parameter	Bemerkung/Beispiel
13	RESET DISK SYSTEM	---	**BDOS**(13)
14	SELECT DISK	E: neues Laufwerk	**BDOS**(14,1)
25	RET.CUR.DISK	---	Funktion **BDOS**(25)
26	SET DMA ADDRESS	DE: Anfang DMA	**BDOS**(26,$C000)
28	WRITE PROTECT DISK	---	Schreibschutz für akt. Laufwerk
37	RESET DRIVE	DE: Laufwerke	**BDOS**(37,3) für A/B
46	GET DISK FREE SPACE	E: Laufwerk	**BDOS**(46,0)
50	DIRECT BIOS CALL	DE: BIOS-Puffer	siehe unten

17.7 Verwendung von BIOS-Funktionen Programm P80.DEM: Lesen eines Sektors, BIOS, BDOS, Register

Mit der oben genannten BDOS-Nummer 50 wird ein sehr mächtiger Aufruf angeboten, der es erlaubt, eine Etage tiefer in die Kiste der BIOS-Routinen zu greifen.

Laut Turbo-Handbuch sollte dies auch mit der Turbo-Anweisung **BIOS** und den Parametern Funktionsnummer, BC-Belegung vonstatten gehen, jedoch waren diesmal unsere Versuche auf diesem Gebiet nicht ganz befriedigend, weil Turbo hier offensichtlich manchmal nicht richtig schaltet, sobald es um das Setzen der DMA-Adresse oder der Bank geht.

Gehen wir deshalb zusätzlich den kleinen Umweg über die BDOS-Funktion Nummer 50, die uns ebenfalls Zugriff auf alle BIOS-Routinen gestattet. Diese sind ebenfalls durchnumeriert, und auch hier geben wir Ihnen nur die an, für die es keinen direkten Zugang von Turbo aus gibt:

BIOS-Nr.	Funktion	Vorbereitung
09	Laufwerk anwählen	Laufwerknummer in C
10	Spur anwählen	Spurnummer in BC
11	Sektor anwählen	Sektornummer in BC
12	Adresse für DMA-Puffer setzen	DMA-Anfang in BC
13	angewählten Sektor lesen	---
14	angewählten Sektor beschreiben	Inhalt ab DMA
25	Blockverschiebung	Anfang alt in DE, neu in HL Anzahl Bytes in BC
29	Bänke setzen für Nr. 25	Quellbank in C, Zielbank in B
28	Speicherbank für DMA setzen	Speicherbanknummer in A

Zur Vorbereitung der BDOS-Funktion Nr. 50 muß ein Puffer bereitgestellt werden, dessen Anfangsadresse als Parameter mitgegeben wird:

Beispiel: **BDOS**(50,$C000)

Bedeutung: Es wird eine BIOS-Funktion aufgerufen, die sich die Parameter aus einem Puffer holt, der ab Adresse $C000 beginnt und in welchem die Werte für die Registerbelegungen in folgender Reihenfolge abgelegt sind:

Beispiel	Parameter
$C000:	BIOS-Funktionsnummer (z.B. 9 für Laufwerk anwählen)
$C001:	Belegung für das A-Register (notwendig bei Nr. 28)
$C002/$C003:	Belegung des BC-Registers (z.B.02/00 = Lo/Hi)
$C004/$C005:	Belegung des DE-Registers (Lo/Hi)
$C006/$C007:	Belegung des HL-Registers (Lo/Hi)

Damit Sie die Leistungsfähigkeit dieser Funktionen erkennen, haben wir uns ein Programm ausgedacht, das wieder nützlicher Natur ist, weil es mit Turbos Befehlssatz allein nicht ohne weiteres denkbar ist: es liest einen Sektor von Diskette in einen Pufferbereich des Arbeitsspeichers und gibt ihn wohlgeordnet auf dem Bildschirm wieder aus. Neu ist hier, daß wir ganz gezielt einen bestimmten Diskettensektor aus einer bestimmten Spur herausgreifen können.

Dazu bietet sich vor allem die Spur 2 an, auf der die Directory-Einträge abgelegt sind, so daß wir auf diese Weise das Inhaltsverzeichnis jederzeit untersuchen können.

Dazu sollten Sie wenigstens grob darüber Bescheid wissen, in welcher Form das Inhaltsverzeichnis auf der Diskette geführt wird:

- Die ersten vier Sektoren (zu je 512 Byte) auf der Spur 2 sind für das Directory reserviert.

- Jeder Datei-Eintrag besteht aus 32 Zeichen, so daß pro Sektor genau 16 Files aufgenommen werden können, also maximal 64 Dateien verwaltet werden.

- Jeder Directory-Eintrag hat folgende Form, wobei uns hier meist nur die ersten 12 Zeichen interessieren:

Byte-Nr.	Bedeutung/Inhalt
0	Laufwerk-Code, Bit 7 gesetzt ===> Datei ungültig
1 bis 8	Dateiname
9 bis 11	Erweiterung des Dateinamens, z.B. PAS oder COM usw.
12	Belegung mit 1: READ ONLY FILE
13	Belegung mit 1: SYSTEM FILE
14 bis 32	Daten für die CP/M-Verwaltung

Im Turbo-Text zu Programm P80.DEM haben wir beide Arten von Aufrufen angeführt. Probieren Sie auch mal die BIOS-Aufrufe, aber bitte mit einer Diskette, von der Sie eine Sicherungskopie haben!

```
      PROGRAM EinenSektorLesen; (* P80.DEM *)
      TYPE
(*A*)   111=Array [0..11] Of Char;
        zeigertyp=^111;
      VAR
        i,j,k:Integer;
        zeiger:zeigertyp;
      BEGIN
        FOR i:=1 TO 512 DO Mem[$D000+i]:=65;
(*B*)   Mem[$D200]:=12;Mem[$D202]:=0;Mem[$D203]:=$D0;
        BDOS(50,$D200);
        {entspricht BIOS(12,$D000);}
(*C*)   Mem[$D200]:=10;Mem[$D202]:=2;Mem[$D203]:=0;
        BDOS(50,$D200);
        {entspricht BIOS(10,2);}
(*D*)   Mem[$D200]:=11;Mem[$D202]:=1;Mem[$d203]:=0;
        BDOS(50,$D200);
        {entspricht BIOS(11,1);}
(*E*)   Mem[$D200]:=13;BDOS(50,$D200);
        {entspricht BIOS(13);}
(*F*)   FOR i:=0 TO 15 DO
        BEGIN
          FOR j:=0 TO 32 DO
          Write(Mem[$D000+i*32+j],' ');
          Write('*** ');
        END;
        WriteLn;
(*G*)   k:=$D000;
        FOR i:=0 TO 15 DO
        BEGIN
          k:=k+32;
          zeiger:=Ptr(k);
          IF Ord(zeiger^[0]) AND 128 =0 THEN
          BEGIN
            FOR j:=1 TO 11 DO
            Write(zeiger^[j]);
            Write('     ');
          END;
        END;
      END.
```

Erläuterungen:

(A) Wir arbeiten mal wieder mit Zeigervariablen. Diesmal mit einem **Array**-Typ, der insgesamt 12 Zeichen faßt.

(B) Wir unterscheiden zwei Puffer: Erstens den Eingabepuffer, der den Sektorinhalt von Diskette aufnehmen soll. Wir lassen ihn bei $D000 beginnen und füllen ihn zunächst mit Code 65, um eine Kontrolle für die richtige Übernahme bei allen Versuchen zu haben.

Zweitens: Als Datenpuffer für die **BDOS**-Aufrufe Nummer 50 verwenden wir die Adressen ab $D200. Für den ersten Aufruf steht dort als erstes die Nummer 12 und für die BC-Register die Belegungen für die DMA-Adresse, die wir mit $D000 festlegen. Ab hier soll der Eingabepuffer beginnen.

(C) Das Anwählen der Spur geschieht in gleicher Weise über die BIOS-Nummer 10. Wir probieren es mal mit Spur 2, der Directory-Spur.

(D) Als Sektor nehmen wir uns mal den zweiten vor, also Sektor 1.

(E) Mit BIOS 13 lesen wir den angewählten Sektor in den angewählten DMA-Puffer ab $D000 ein.

(F) Zunächst lassen wir uns den Code des gesamten Sektors ausgeben und drucken als Trennzeichen zwischen den einzelnen Einträgen einer Datei drei Sternchen.

(G) In einer zweiten Schleife lassen wir uns dann den Code übersetzt in die ASCII-Zeichen ausgeben und untersuchen dabei jeweils das erste Zeichen auf ein gesetztes höchstes Bit. Steht dies auf Null, dann geben wir von dieser gültigen Datei den achtstelligen Namen und die dreistellige Erweiterung aus.

Vorsicht mit BIOS-Aufrufen:

Wir haben die Erfahrung gemacht, daß die direkten BIOS-Aufrufe zu Fehlern führen, insbesondere die Auswahl von Spur und Sektor spricht nicht richtig an. Bei den Versuchen mit **BIOS**(13) entstanden sonderbare Disketteninhalte! Der sichere Weg geht somit über BDOS-Aufrufe.

17.8 Externe Maschinenprogramme Programm P81.DIR: External, BlockRead, Puffer, Dateien

Turbo ist in der Lage, Maschinenprogramme in das Hauptprogramm einzubinden, wenn sie sich außerhalb des Turbo-Textes befinden. Voraussetzung ist allerdings, daß sich der Maschinencode bereits im Arbeitsspeicher befindet.

Aufgerufen werden solche externen Maschinenprogramme mit dem Wort **External** und der Adresse dahinter, an der der Maschinencode beginnt.

Die CP/M-Versionen von Turbo besitzen zwar nicht von Haus aus die Fähigkeit, auf Diskette ausgelagerte **External**s zu verarbeiten, aber dem kann man mit einer kleinen Zusatzprozedur abhelfen.

Das nächste Programm P81.DIR ist etwas komplexer und erfordert viel Aufmerksamkeit und besondere Sorgfalt beim Programmieren, weil sonst die üblichen Abstürze gehäuft auftreten. Wir wollen nämlich diesmal ebenfalls die Diskettensektoren der Spur 2, also das Inhaltsverzeichnis, lesen. Jedoch soll dies ein **External**-Programm übernehmen, das auf Diskette bereitgehalten wird.

Damit schlagen wir gleich zwei Fliegen mit einer Klappe: Wir erstellen uns ein nützliches Dienstprogramm, das sich überall als **Include**-Procedure oder als eigenständiges Programm aufrufen läßt und lernen gleichzeitig, wie man mit Maschinenprogrammen umgeht. Uns kommt es diesmal überhaupt nicht darauf an, daß wir elegante Programmierschritte unternehmen, sondern vielmehr auf das Verständnis der Zusammenhänge. Also versuchen wir, mit möglichst einfachen Mitteln zum Ziel zu kommen.

Der einfachere Teil ist das Turbo-Programm in der bisher schon gewohnten Weise. Wir stellen es unter dem Namen »Directory_Lesen« mit allen Einzelprozeduren noch einmal komplett vor. Es soll folgende Aufgaben übernehmen:

- Der Maschinencode einer auf Diskette vorhandenen Routine wird in einen freien Bereich des Turbo-Arbeitsspeichers übernommen. Damit wird eine aufrufbare **External**-Prozedur angelegt.

- Aufrufen einer **External**-Prozedur, die bestimmte Sektoren von der Diskette lesen und in einem bereitgestellten Puffer ablegen kann.

- Ausgeben der in dem Puffer stehenden Daten in einer entsprechenden Form.

Hier eine Möglichkeit, wie man diese Aufgaben lösen kann:

```
      PROGRAM Directory_Lesen; (* P81.DIR *)
      TYPE
        name=String[12];
      VAR
        sektor:Integer;

(*A*) PROCEDURE LiesSektor(spur,sektor:Integer);External $CF00;

(*B*) PROCEDURE LiesPuffer;
      TYPE
        111=Array [0..11] Of Char;
        zeigertyp=^111;
      VAR
        i,j,k:Integer;
        zeiger:zeigertyp;
      BEGIN
      {Der Pufferanfang $D000 ist
           im externen Maschinenprogramm festgelegt}
        k:=$D000-32;
        FOR i:=0 TO 15 DO
        BEGIN
          k:=k+32;
          zeiger:=Ptr(k);
(*C*)     IF((Ord(zeiger^[0]) AND 128=0) AND (zeiger^[1]<>#255))THEN
          BEGIN
            FOR j:=1 TO 11 DO
            Write(zeiger^[j]);
            Write('     ');
          END;
        END;
      END;

(*D*) PROCEDURE ExCodeIn(datname:name);
      VAR
        datei:File;
(*E*)   absfeld:Char Absolute $CF00;
      BEGIN
        Assign(datei,datname);
        Reset(datei);
(*F*)   BlockRead(datei,absfeld,1);
        Close(datei);
      END;
```

```
(*G*) PROCEDURE Vorbereitung;
      VAR i:Integer;
      BEGIN
        ClrScr;
        FOR i:=1 TO 512 DO Mem[$D000+i]:=0;
        ExCodeIn('SEKTOR.EXT');
      END;

      BEGIN (* Hauptprogramm *)
(*H*)   Vorbereitung;
        FOR sektor:=0 TO 3 DO
        BEGIN
          LiesSektor(2,sektor);
          LiesPuffer;
        END;
      END.
```

Erläuterungen:

(A) Die Prozedur »LiesSektor« hat auch noch zwei Parameter vom Typ Integer, mit denen wir die gewünschte Spur und den gewünschten Sektor anwählen können. Der Aufruf dieser **External**-Prozedur liegt bei Adresse $CF00. Dorthin muß also der Maschinencode transferiert werden.

(B) Die Prozedur »LiesPuffer« sollte Ihnen schon aus dem vorigen Beispielprogramm P80.DEM bekannt vorkommen. Wir verzichten aber diesmal auf die Ausgabe der Codezahlen, sondern konzentrieren uns ganz auf den Ausdruck der Dateinamen, weil wir ja das Inhaltsverzeichnis sehen wollen.

In unserem Beispiel liegt der Eingabepuffer fest bei $D000, weil er vom Maschinenprogramm aus ab dieser Anfangsadresse gefüllt wird. Man könnte der **External**-Prozedur »LiesSektor« auch noch einen weiteren Parameter für diesen Pufferanfang mitgeben, um ihn variabel zu halten.

(C) Wir sortieren mit **AND** 128 im ersten Zeichen jedes Eintrags die nicht aktuellen Dateien aus und lassen uns auch die Leerräume nicht ausgeben, die mit Code 255 gefüllt sind.

(D) Mit der Prozedur »ExCodeIn« erstellen wir uns ein Werkzeug, mit dem man das auf Diskette unter einem bestimmten Dateinamen »datname« vorhandene Maschinenprogramm in den Arbeitsspeicher hereinholen kann.

(E) Die Anfangsadresse muß dabei die gleiche sein, die hinter der **External**-Prozedur steht. Wir bedienen uns dazu einer neuen Variablenart. Sie ist vom Typ **Absolute** und liegt an der Stelle, die wir mit einer Integerzahl angeben. Dabei kann es sich um einfache oder um Zeigervariablen handeln. Wir bescheiden uns mit dem Typ **Char**, weil es uns nur auf die Anfangsadresse ankommt.

(F) Nachdem unser Maschinenprogramm kürzer ist als 128 Zeichen, genügt es, mit **BlockRead** einen einzigen Block einzulesen und ihn direkt an die geforderte Stelle zu übertragen. Das spart enorm Zeit.

(G) Zur Vorbereitung löschen wir den Schirm und einen eventuellen Pufferinhalt und laden das externe Maschinenprogramm, das auf der Diskette unter dem Namen »SEKTOR.EXT« zu finden ist, mit »ExCodeIn« an die richtige Stelle.

(H) Das Hauptprogramm beginnt konsequenterweise mit den Vorbereitungen und liest dann Sektor für Sektor aus dem Inhaltsverzeichnis auf Spur 2 und druckt die Dateinamen aus. Das Ganze dauert ca. 3 Sekunden (nach dem Kompilieren), wobei die meiste Zeit auf das Anlaufen des Diskettenlaufwerks entfällt.

Jetzt bleibt aber hoffentlich eine Frage offen: Wie erzeugt man das Maschinenprogramm, das hier von Diskette unter dem Namen SEKTOR.EXT abgerufen wird? Lesen Sie dazu den nächsten Abschnitt, in dem wir gezwungenermaßen etwas tiefer in die Programmiergeheimnisse Ihres CPC 6128 hineinleuchten müssen. Sollte Ihnen das über den Kopf wachsen, was nun mit der Programmierung in Maschinensprache folgt, dann wenden Sie die entstandenen Prozeduren einfach an, ohne sich über den Aufbau Gedanken zu machen. Auch das ist mit Turbo ohne weiteres möglich, da man beim Einbinden von vorgegebenen Routinen in die eigenen Programme lediglich auf eventuelle Parameterübereinstimmung und globale Deklarierungen achten muß.

17.8.1 Erstellen von Assembler-Programmteilen Turbo als Assembler-Editor

Maschinensprache besteht nur aus Codezahlen, mit denen der Prozessor (hier der Z80A) gefüttert wird. Nun braucht man sich aber nicht mühsam für jeden Maschinenbefehl die Codierung aus Tabellen zusammenzuklauben, sondern schreibt die entsprechenden Befehle in Form eines Textes, den man einem Assembler-Programm überreicht, welches daraus die Maschinenbefehle erzeugt.

Noch komfortabler arbeiten sogenannte Makro-Assembler, die in einer Art Unterprogrammtechnik komplexe Einheiten zu Makros zusammenfassen und sie an den richtigen Stellen im Maschinencode plazieren.

Einige Assembler oder Makro-Assembler haben meist kein eigenes integriertes »Textprogramm«, sondern erwarten, daß der zu assemblierende Text auf Diskette oder ähnlichem bereitgehalten wird.

Was hat das mit Turbo zu tun?

Nun, wir können den superkomfortablen Turbo-Editor, mit dem wir bisher unseren Turbo-Programmtext erstellt haben, genausogut dafür einsetzen, einen Assembler-Quelltext zu schreiben, und brauchen uns nicht mit dem etwas eigenartigen ED-Editor aus dem CP/M-System herumzuplagen.

An Hand des nächsten Beispiels sollten Sie das einmal nachvollziehen, was wir Ihnen gleich bieten, auch wenn Sie vielleicht (noch) gar nicht an Assembler interessiert sind.

Den folgenden Assemblertext können Sie mit Turbo eintippen und abspeichern. Wir erzeugen daraus das externe Maschinenprogramm SEKTOR.EXT, das im Programm P81.DIR benötigt wird. Es liest einen bestimmten Sektor von Diskette und legt die Daten in einem Speicherbereich ab, an den Turbo zur Weiterverarbeitung ohne weiteres herankommt.

```
; ******************* P82.ASM Lesen/Ablegen eines Sektors *******
; (A)
    RET
    RET
    RET
; (B) **************** Anfang des Maschinenprogramms *************
    DI            ; Interrupts verhindern
    EXX           ; Registerinhalte in zweiten Registersatz retten
    POP HL        ; Return-Adresse vom Stack holen
    POP BC        ; 2. Parameter (Sektornummer) vom Stack
    POP DE        ; 1. Parameter (Spurnummer) vom Stack
    PUSH HL       ; Return-Adresse wieder ablegen
    EX (SP),HL    ; Stackpointer nach HL
    PUSH HL       ; .. und retten
    PUSH BC       ; Sektornummer vorläufig ablegen
    PUSH DE       ; Spurnummer ebenfalls
    LD A,193      ; Byte für Bank 1 (192+1) laden
    LD BC,#7FC0   ; und über BC
    OUT (C),A     ; an Gate Array senden (Bankswitching auf Bank 1)
```

```
    LD BC,#D000  ; DMA-Anfang #D000 laden
    CALL #FC24   ; und mit BIOS 12 setzen
    POP BC       ; Spurnummer vom Stack holen
    CALL #FC1E   ; und mit BIOS 10 anwählen
    POP BC       ; Sektornummer vom Stack holen
    CALL #FC21   ; und mit BIOS 11 anwählen
    CALL #FC27   ; angewählten Sektor mit BIOS 13 nach DMA
    LD A,194     ; Bank 2 (Turbo-Bank)
    LD BC,#7FC0  ; über BC
    OUT (C),A    ; einschalten
    POP HL       ; Stackpointer wieder vom Stack holen
    EX (SP),HL   ; und auf ursprünglichen Wert setzen
    EXX          ; gerettete Registerinhalte wieder einsetzen
    EI           ; Interrupts zulassen
    RET          ; Rücksprung zum aufrufenden Programm
;   (C) ********** Ende des eigentlichen Maschinenprogramms ******
    RET
    RET          ; Abschlußcode (willkürlich)
*****************************************************************
```

Die Kommentare hinter den Strichpunkten brauchen Sie nicht mit einzutippen, sie dienen nur dem Verständnis.

Einige Erläuterungen sind sicher noch nützlich:

- Die Sprungtabelle für die BIOS-Funktionen befindet sich im Speicherblock 7, beginnend mit der Basisadresse $FC00. Wenn Sie die Nummer des BIOS-Aufrufes mit 3 multiplizieren und das Ergebnis zu $FC00 addieren, erhalten Sie die direkte Einsprungadresse der gewünschten BIOS-Funktion.

- Die BIOS-Routinen selbst liegen in anderen Speicherblöcken der Bank 1. Das erfordert ein Bankswitching, um sie anlaufen zu können. Der Puffer, dessen Anfang wir hier mit $D000 gewählt haben, liegt im Schnittbereich von Bank 1 und 2, so daß wir allein für das Ablegen der Daten nicht eigens umschalten müßten.

  Sollten Sie für diesen Datenpuffer (DMA) aber einen geschützten Bereich unterhalb des Turbo-Codes ausgesucht haben (siehe dazu 17.8.6), dann ist dieses Verfahren unumgänglich.

- Zu Beginn (A) und am Ende (C) des Assemblerprogramms finden Sie jeweils dreimal den Befehl RET. Dies dient uns später zum Auffinden des Anfangs des eigentlichen Maschinencodes, der bei (B) beginnt (siehe unten).

Sie sollten nun diesen Assembler-Quelltext abspeichern. Er ist dann unter »P82.ASM« auf Diskette zu finden und mit der Erweiterung ASM weiterzuverarbeiten. Auf dieser Diskette sollte auch ein Assembler sein, wie z.B. ASM.COM oder MAC.COM (von den Systemdisketten Ihres Schneider CPC).

Steigen Sie nun mit **<Q>** über das Hauptmenü aus Turbo aus und starten Sie Ihr Assemblerprogramm unter CP/M mit ASM P82 oder dem entsprechenden Befehl Ihres Assemblers.

Jetzt wird der Quelltext »P82.ASM« von der Diskette geholt. Daraus werden mehrere Dateien erzeugt, die Sie sich mit TYPE anschauen können. Je nach Assemblertyp finden Sie die Files »P82.PRN«, »P82.HEX« oder ähnliche vor. Auch von Turbo aus lassen sich deren Inhalte auf dem Schirm oder dem Drucker ausgeben.

Wurde ohne Fehler assembliert, kann der nächste Schritt erfolgen: Aus der Datei »P82.HEX«, die lediglich den Assemblercode in Form von zweistelligen Hexzahlen im ASCII-Code aufbereitet hat, muß der Maschinencode hergestellt werden. Das kann durch den Aufruf eines Hilfsprogramms wie HEXCOM erfolgen. Sie können sich aber auch eine Turbo-Routine schreiben, die aus jeweils zwei Hexziffern den entsprechenden Bytewert erstellt und auf Diskette schreibt. Wie man zu den Hexziffern kommt, haben wir Ihnen mit Programm P76.PAS vorgeführt. Die Umkehrung dazu sollte Ihnen selbst gelingen.

Verwenden Sie eine Umformung aus Ihrem Assemblerpaket (z.B. HEXCOM.COM), dann erhalten Sie normalerweise ein lauffähiges Maschinenprogramm, das in unserem Fall unter dem Namen »P82.COM« abgespeichert wird. In diesem sogenannten Objektcode sind aber noch einige Zusatzanweisungen zu Beginn und am Ende enthalten, die wir für die Verwendung als **External**-Prozedur nicht gebrauchen können.

Bereinigen wir also noch diesen Objektcode, so daß mit dem ersten Byte auch wirklich der Anfang der externen Prozedur vorliegt. Und dazu bedienen wir uns der eingangs erwähnten dreifachen RET-Anweisungen, die ansonsten wohl nie in einem normalen Programm auftreten.

Anmerkung: Den Maschinencode von SEKTOR.EXT finden Sie auch im Listing des Programms »FileHand.Bon«, das wir Ihnen als Bonbon gestiftet haben (Anhang A).

17.8.2 Bereinigen des Objektcodes Programm P83: offene Dateien, File

Das folgende Turbo-Programm entfernt den Vorspann aus einem Objektcode, der durch den Assembler erzeugt worden ist. Voraussetzung dazu ist, daß wir den Anfang des Maschinenprogramms einwandfrei erkennbar markiert haben. Dazu haben wir in den Quelltext dreimal RET geschrieben, was mit dem Wert 201 oder $C9 codiert wird.

Der Aufbau des Bereinigungsprogramms P83 »ObjCheck« ist relativ einfach:

(A) Abfrage des Dateinamens der zu bereinigenden Objekt-Datei. In unserem Fall wäre das z.B. »P82.COM«.

(B) Eingabe eines Dateinamens, unter dem die bereinigte Datei angelegt werden soll (z.B. »SEKTOR.EXT«).

(C) Eröffnen der vorgegebenen Datei und Eröffnen der zweiten neuen Datei.

Es ist auch in Turbo möglich, mehrere Dateien gleichzeitig geöffnet zu halten.

(D) Einlesen eines ganzen Datenblocks in ein eindimensionales Feld. Dabei gehen wir von einem Codeblock aus, der nicht mehr als 128 Zeichen umfaßt. Für größere Maschinenteile muß diese Stelle modifiziert werden. Um zu erkunden, wieviele Blöcke notwendig sind, können Sie zunächst die Bytes abzählen lassen, bis zum zweiten Mal eine Folge von drei hintereinander liegenden RETs (der Wert 201) auftritt.

(E) Durchsuchen der Objektdatei, bis zum ersten Mal dreimal der Wert 201 auftaucht. Die Nummer, bei der das erste Byte des externen Programms liegt, halten wir in der Variablen »erster« fest.

(F) Lassen wir uns den Code des bereinigten Maschinenprogramms auf dem Schirm ausgeben, dann erkennen wir das Ende am nochmaligen dreifachen Auftreten der Zahl 201.

(G) Mit einem einzigen **BlockWrite** übertragen wir einen Block, beginnend mit dem Element »erster« auf die Diskette. Dabei werden auch noch etliche Überhang-Bytes mitgenommen, die im Speicher hinter dem Maschinencode stehen, was uns aber nicht zu interessieren braucht, weil mit dem ersten RET sowieso ein Rücksprung zum aufrufenden Programm erfolgt.

(F) Nach der letzten Übertragung werden beide Dateien ordnungsgemäß geschlossen.

```
        PROGRAM ObjCheck;   (* P83 *)
        VAR
          datei1,datei2:String[12];
          file1,file2:File;
          code:Char;
          feld:Array [1..128] Of Char;
          z,i,letzter,erster:Integer;
        BEGIN
          ClrScr;z:=0;
(*A*)     Write('Dateiname(Obj.-Datei): ');
          ReadLn(datei1);
(*B*)     Write('Dateiname(bereinigt):  ');
          ReadLn(datei2);
(*C*)     Assign(file1,datei1);
          Assign(file2,datei2);
          Reset(file1);
          Rewrite(file2);
(*D*)     BlockRead(file1,feld,1);
          FOR i:=1 TO 128 DO
          Write(Ord(feld[i]),' ');
          z:=0;i:=0;
(*E*)     REPEAT
            i:=i+1;
            If Ord(feld[i])=201 THEN z:=z+1
            ELSE z:=0;
          UNTIL z=3;
          erster:=i+1;
(*F*)     FOR i:=erster TO 128  DO Write(Ord(feld[i]),' ');
(*G*)     BlockWrite(file2,feld[erster],1);
          Reset(file2);
          BlockRead(file2,feld,1);
          WriteLn;
          FOR i:=1 TO Ord(feld[1]) DO Write('*',Ord(feld[i]):4);
(*H*)     Close(file1); Close(file2);
        END.
```

Weitere Erläuterungen sollten sich eigentlich erübrigen, wir haben in den vorhergehenden Kapiteln alle Punkte geklärt. Schauen Sie im Zweifelsfall dort noch einmal nach. Neu ist hier lediglich, daß wir mit zwei gleichzeitig geöffneten Dateien des Typs **File** arbeiten.

Zusammenfassung:

Mit Turbo läßt sich komfortabel ein Assembler-Quelltext erstellen. Das Assemblieren kann mit einem Assembler erfolgen, dessen Objektcode aber für die Verwendung in Turbo-Programmen erst bereinigt werden muß.

In unserem Beispiel ergeben sich auf Diskette vier Dateien:

– der Quelltext »P82.ASM«
– der sogenannte INTEL-Hex-Code »P82.HEX«
– eine Objektcode-Datei z.B. »P82.COM« oder »P82.OBJ« oder ähnliches
– das reine Maschinenprogramm »SEKTOR.EXT"

Anmerkung: Sie können natürlich auch mit dem CP/M-eigenen SID-Programm arbeiten, das Ihnen Assemblieren und Disassemblieren gestattet. Die Vorteile eines guten Assemblers besitzt es zwar nicht, aber SID eignet sich durchaus für kleine Maschinenteile. Die Beschreibung dazu sollten Sie sich aus den CP/M-Unterlagen holen. Wir interessieren uns hier nur für die Möglichkeiten, die Turbo bietet.

Übrigens: Sie sollten sich Abschnitt 17.9.4 genauer ansehen, wenn es Sie interessiert, wie man externe Unterprogramme ohne Zuhilfenahme einer Diskettendatei zum Einsatz bringen kann.

17.8.3 External-Unterprogramme mit Festwertparametern Parameterübergabe am Beispiel »SEKTOR.EXT«

Wie Sie am vorigen Beispiel gesehen haben, ist auch bei den **External**-Prozeduren (und -Funktionen) eine Parameterübergabe möglich. Die Form entspricht genau der, wie wir sie bereits in den ersten Kapiteln besprochen haben.

Allerdings muß man sich einige Kenntnisse aneignen, damit man weiß, wie diese Parameter von dem aufgerufenen Maschinenprogramm verarbeitet werden. Schauen wir uns das zunächst einmal bei der einfacheren Art, nämlich bei den Festwertparametern an, indem wir die Zusammenhänge zwischen dem Prozedur-Aufruf und dem Ablauf des Programms »SEKTOR.EXT« genauer untersuchen.

Der Aufruf heißt in unserem Beispiel:

```
liessektor(spur,sektor);
```
(»spur« und »sektor« sind Integerzahlen)

Sobald das Turbo-Programm auf diesen Aufruf stößt, legt es die Parameter »spur« und »sektor« als Integerzahlen in jeweils zwei Byte auf den Stack, und zwar in der Reihenfolge wie sie anfallen: »spur« wird als erstes abgelegt, »sektor« als nächstes.

Die Festwertparameter beanspruchen auf dem Stack genau so viele Bytes, wie es ihrem Typ entspricht. Ausnahme sind die strukturierten Typen, bei denen die Anfangsadresse abgelegt wird:

– **Integer**zahlen, **Boolean**, **Char**, **Bytes**, sonstige Skalare: 2 Byte
– Reelle Zahlen (Typ **Real**): 6 Byte
– **Strings**: für die Länge 1 Byte + Stringlänge
– Mengen: 32 Byte
– Zeiger: 2 Byte
– **Arrays, Records**: 2 Byte (Anfangsadresse)

Damit Turbo (genauer das von Turbo erzeugte Kompilat) nach Durchlaufen des Unterprogramms wieder hinter das aufrufende Programm zurückfindet, wird abschließend die Rücksprungadresse auf den Stack gelegt. Beim Eintreten in die Maschinenroutine sieht der Stapel, der in Richtung der niedrigeren Adressen aufgefüllt wird (also von oben nach unten), folgendermaßen aus:

... alte Stackwerte ...		?? uninteressant
Spurnummer:	Hi Lo	2 Byte (1 Wort)
Sektornummer:	Hi Lo	2 Byte (1 Wort)
Rücksprung:	Hi Lo	2 Byte (1 Wort)
		← Stackpointer

Bild 17-3: Stack beim Eintreten in die Maschinenroutine

Wichtig ist nun, daß der Programmierer den Stack selbst verwaltet und bereinigt, wenn er mit **External**-Unterprogrammen arbeitet. Dazu gehören folgende Schritte:

- Die Rücksprungadresse wird vom Stack abgehoben. Damit steht der Stackpointer 2 Byte weiter oben, weil POP-Operationen immer mit einem Wort arbeiten.

- Jetzt kommt man an die Parameter heran, die auf die gleiche Weise vom Stack entfernt und in Register oder sonstige freie Speicherstellen gerettet werden.

- Die Rücksprungadresse wird wieder auf den Stack gelegt und die Stellung des Stackpointers wird in dieser Lage ebenfalls gerettet und auf den Stack gelegt.

- Im Laufe des Maschinenprogramms können dann mit PUSH und POP beliebig oft Daten auf dem Stack hin- und hergeschoben werden. Aber nun heißt es aufpassen, daß nach Beendigung des Unterprogramms der gleiche Zustand wiederhergestellt wird wie zu Beginn. Lediglich die Rücksprungadresse sollte noch auf dem ursprünglichen Stack liegen. Was dann noch kommt, ist uninteressant, wenn der gerettete Stackpointer wieder eingesetzt wird. Dann ist das Programm wieder zum Rücksprung an die richtige Stelle bereit.

- Beließe man die Parameter auf dem Stack, dann würde er durch mehrfache Aufrufe immer weiter anwachsen und eventuell in andere Speicherbereiche hineinlaufen, was in den meisten Fällen einen Absturz des Rechners zur Folge hat.

Übrigens braucht man sich bei **Inline**-Programmen über die Stackverwaltung keine Gedanken machen. Dort wird der Stack auch bei der Verwendung von Parametern automatisch bereinigt. Im nächsten Abschnitt werden wir das noch einmal anschaulich demonstrieren.

17.8.4 External-Prozeduren mit Variablenparametern Programm P84 mit External LO85.EXT und LO85.INL

Wenn einem **External**-Unterprogramm Wechselwert-(Variablen-)Parameter mitgegeben werden, dann wird grundsätzlich zuerst die Adresse auf den Stack gelegt, wo das erste Byte des Variablenparameters zu finden ist. Der Typ spielt dabei keine Rolle. Nach dieser Variablen-Anfangsadresse legt Turbo dann wieder die Rücksprungadresse darüber (von der Speichernumerierung aus betrachtet eigentlich darunter, weil der Stack mit der Spitze nach unten »hängt«.)

Der Programmierer hat dafür zu sorgen, daß der Variableninhalt innerhalb des Maschinenprogramms wie gewünscht verarbeitet wird.

Das Demonstrationsprogramm P84.DEM soll uns nun die Arbeitsweise von derartigen **External**-Prozeduren vorführen. Dabei bedienen wir uns der im Programm P81.DIR schon angewendeten Methode, das **External** mit Hilfe der Prozedur »ExCodeIn« von Diskette einzulesen.

```
      PROGRAM External_VAR_Parameter_Demo;   (* P84.DEM *)
      TYPE name=String[12];
      VAR
(*A*)   intzahl:Integer;

(*B*) PROCEDURE lowert (VAR zahl:Integer);External $CF00;

(*Z*) PROCEDURE loin (VAR zahl:Integer); (* LO85.INL *)
      BEGIN
        Inline($2A/zahl/$23/$36/0);
      END;

(*C*) PROCEDURE ExCodeIn(datname:name);
      VAR
        datei:File;
        absfeld:Char Absolute $CF00;
      BEGIN
        Assign(datei,datname);
        Reset(datei);
        BlockRead(datei,absfeld,1);
        Close(datei);
      END;

(*D*) PROCEDURE vorbereitung;
      BEGIN
        ClrScr;WriteLn(#27,'p','External-Prozedur mit
                       VAR-Parameter',^J^M,#27,'q');
        ExCodeIn('LO85.EXT');
      END;
      BEGIN
(*E*)   vorbereitung;
        Write('Belegung der Variablen »intzahl«: ');
        ReadLn(intzahl);
        lowert(intzahl);
        {loin(intzahl);}
        WriteLn('Lo-Wert von »intzahl«:  ',intzahl);
        Write('Belegung von »intzahl«: ',intzahl);
      END.
```

Erläuterungen:

(A) »intzahl« heißt die zu behandelnde Integerzahl.

(B) Die Prozedur »lowert« bestimmt von der Integerzahl den Wert des Lo-Bytes, berücksichtigt also das Hi-Byte nicht. Das entspricht der Standardfunktion **Lo**.

(C) »ExCodeIn« legt den eingelesenen Maschinencode ab Adresse $CF00 im Speicher ab.

(D) Eine Vorbereitungsprozedur erledigt u.a. das Einrichten der **External**-Routine.

(E) Zur Demonstration belegen wir »intzahl« mit einem Wert aus dem Integerbereich bis 32767. Der Aufruf von »lowert« bearbeitet die Zahl und wir können uns davon überzeugen, daß sich der Variableninhalt dadurch verändert hat.

Das zugehörige Maschinenprogramm ist von recht bescheidenem Umfang:

```
; ****************** EXTERNAL LO85.ASM ***************************
;
    POP BC        ; Rücksprungadresse nach BC holen
    POP HL        ; Adresse der Integerzahl nach HL holen
    PUSH BC       ; Rücksprungadresse wieder auf den Stack legen,
                  ; der damit bereinigt ist.
    INC HL        ; die in HL enthaltene Adresse um 1 erhöhen, dann
                  ; zeigt HL auf das Hi-Byte der Integerzahl
    LD (HL),0     ; dieses Hi-Byte auf Null setzen
    RET           ; Rücksprung zu der im Stack als letztes fest-
                  ; gehaltenen Adresse
; ****************************************************************
```

Hier ist es nicht notwendig, den Stackpointer zu retten, weil weder auf andere Banks umgeschaltet, noch andere Routinen aufgerufen werden, die den Stackzeiger »verbiegen« könnten.

Der Unterschied zur Verwendung von Festwertparametern ist der, daß man mit indirekten Operationen, wie z.B. LD (HL),0 die Inhalte der Variablen verändern muß.

Von großer praktischer Bedeutung ist unser **External**-Beispiel nicht. Es soll auch nur die Arbeitsweise dieser Unterprogrammart verdeutlichen. Damit Sie auch einen Vergleich zu den **Inline**- Unterprogrammen haben, führen wir die gleiche Operation mit folgender Prozedur durch, die wir in unser Demo-Programm P84.DEM bei (Z) bereits eingefügt haben:

```
(*Z*)  PROCEDURE loin (VAR zahl:Integer); (* LO85.INL *)
       BEGIN
         Inline($2A/zahl/     { LD HL,(zahl) }
                $23/          { INC HL       }
                $36/0);       { LD (HL),0    }
       END;
```

Wie Sie aus dem einwandfreien Arbeiten dieser Prozedur sehen, übernimmt Turbo die Parameter- und Stackbehandlung, so daß der Programmierer davon entlastet ist. Bei externen Unterprogrammen dagegen muß man sich um all das selbst kümmern.

17.8.5 External-Funktionen Programm P86.DEM mit External LO87.EXT

Zur Zusammenfassung unserer Überlegungen, wie Turbo mit Unterprogrammen umgeht, lösen wir das gleiche Problem, nämlich die Ausgabe des Lo-Werts einer Integerzahl noch einmal. Diesmal aber mit einer **External**-Funktion.

Dazu bauen wir das Demo-Programm P84.DEM an einigen Stellen um und nennen es P86.DEM. Hier nur die abgeänderten Teile:

```
        PROGRAM External_Funktion_Demo; (* P86.DEM *)
        ...
(*A*)   FUNCTION lowert (zahl:Integer): Integer; External $CF00;
        ...
(*F*)   ...
        'External-Funktion mit Festwert-Parameter',...
        ExCodeIn('LO87.EXT');
...
(*G*)   ...
        Write(lowert(intzahl));
        WriteLn(' ist Lo-Wert von »intzahl«:  ');
        ...
```

Erläuterungen:

(A) Diesmal verwenden wir den Festwertparameter »zahl«, so daß der Inhalt der Übergabevariablen »intzahl« nach Rückkehr von der Funktion unverändert bleibt. Auch das Funktionsergebnis ist vom Typ **Integer**.

(F) Wir benötigen auf Diskette eine neue Maschinenroutine, die als Funktion arbeitet.

(G) Beachten Sie, daß bei Aufrufen von Funktionen das Funktionsergebnis gleich unter dem Funktionsbezeichner weiterverarbeitbar ist, während Prozeduren niemals ihr Ergebnis unter ihrem Prozedurnamen ablegen.

Das notwendige Maschinenprogramm LO87.EXT muß nun etwas abgeändert werden. Dazu die notwendigen Grundlagen:

Zum Schluß des Maschinenprogramms müssen die Funktionsergebnisse in einer von Turbo festgeschriebenen Art abgelegt werden:

- Integerergebnisse, Zeiger und skalare Typen, die nicht mehr als zwei Byte benötigen, müssen im Registerpaar HL (Hi/Lo) abgegeben werden.

- Reelle Zahlen haben in den Registern BC,DE und HL zu stehen, wobei L den Exponenten aufnimmt und B das höchstwertige Byte.

- Bei Strings und Mengen reichen die Register nicht mehr aus. Deswegen wird der Stack als Puffer benützt. Bei Stringergebnissen muß das letzte Byte auf dem Stack die Stringlänge beinhalten, und genau so viele Bytes müssen auf dem Stack darunter-gelegt worden sein, wie der String lang ist. In diesem Fall darf also am Schluß der Routine der Stack nicht bereinigt werden.

In unserem einfachen **Integer**-Fall brauchen wir also nur dafür zu sorgen, daß bei dem Wert des Parameters das Hi-Byte mit Null belegt wird und daß der neue Zahlenwert auch vor dem Rücksprung in HL enthalten ist.

```
; ****************** EXTERNAL-Funktion LO87.EXT ******************
;
   POP BC          ; Rücksprungadresse retten
   POP HL          ; Integerparameter holen (Festwert)
   PUSH BC         ; Stackbereinigung abschließen
   LD H,0          ; Hi-Byte löschen
   RET             ; Rücksprung, Ergebnis ist bereits in HL
; ****************************************************************
```

Übrigens: Inline-Funktionen mit einer Ergebnisübergabe aus den Registern HL klappen nicht, da Turbo offensichtlich dieses Registerpaar für seine Rückorientierung braucht.

17.8.6 Zusammenfassung und Ergänzungen zu External-Unterprogrammen

- **External**-Unterprogramme können als Funktionen oder Prozeduren aufgebaut sein. Beide Arten können mit beiden Arten von Parametern (Festwert- oder Variablen-parameter) arbeiten.

- Der Maschinencode muß vor dem Kompilieren im Arbeitsspeicher bereitgehalten werden. Die Anfangsadresse des Codes wird beim Aufruf des Unterprogramms mitgeteilt.

- Als Speicherbereich für **External**-Programme ist ein freier Raum zu wählen, der weder vom Turbo-Text noch vom Kompilat, dem Heap oder den Stacks erreicht werden kann.

  Auf einfache Weise findet man einen unbenützten Speicherbereich, wenn man sich im Haupt-Menü von Turbo die Anfangs- und Endadressen betrachtet: Der Turbo-Text steht normalerweise ab Adresse $819B. Wieviele Bytes noch frei sind, wird darunter angezeigt. Sobald Sie aber mit der Voreinstellung »M« Ihren Turbo-Quelltext compilieren oder auch starten, wird das Kompilat oberhalb vom Turbo-Text abgelegt. Das kann eventuell schon Überschneidungen mit einem **External** geben.

  Eine dauerhafte, sichere Lösung für Programme, die **Externals** verwenden, bietet sich darin an, daß man die Startadresse für das Kompilat verändert, also um den benötigten Bereich erhöht. Dann kann man zwischen der Turbo-Library, die bis $20E1 reicht, und dem Beginn des Kompilat-Codes bequem und geschützt ein externes Unterprogramm oder absolute Variablen verstauen.

  So könnten in unseren Beispielen die **Externals** z.B. bei $2100 abgespeichert werden. Dieser Bereich ist aber nur dann nutzbar, wenn der Speicher frei ist von Turbo-Editor und -Compiler. Und das ist er nicht, wenn Turbo geladen ist. Versuchen Sie also nicht, mit der voreingestellten Memory-Kompilierung ein derartiges Programm laufen zu lassen.

Schritt für Schritt:

1. Turbo-Quelltext schreiben.

2. Aus dem Hauptmenü **<O>** für Compiler-Optionen wählen.

3. Mit **<C>** die Kompilierart wählen, die ein lauffähiges Programm auf Diskette mit dem Zusatz .COM erzeugt.

4. Die Startadresse mit **<S>** und der Eingabe eines neuen, erhöhten Werts nach oben setzen.

5. Mit <Q> zurück ins Hauptprogramm springen.

6. Und nun <C> für Kompilieren drücken.

7. Wenn Sie Ihr Programm jetzt ausprobieren möchten, tun Sie das mit <X> für Execute. Es ist nun aber auch ohne Turbo-Programm unter CP/M für sich allein lauffähig, weil Turbos COM-Kompilate immer den notwendigen Befehlssatz (Library) mit sich führt. Beim Laden von Diskette wird der Code immer ab der gewählten Startadresse liegen und weder mit CP/M noch dem darunterliegenden **External** in Berührung kommen (wenn Sie nichts falsch gemacht haben).

17.9 External-Anwendungen

Zum Abschluß unseres Ausflugs in die Ebene der Maschinensprache zeigen wir noch in zwei nützlichen Anwendungen, wie man mit Turbo-Programmen einen Bildschirminhalt auf Diskette und zurück bewegen kann. Das ist vor allem für diejenigen interessant, die sich näher mit den Grafikfähigkeiten ihres Rechners befassen und ihre Produkte als Bildschirmseiten aufbewahren wollen.

17.9.1 Bildschirmverwaltung des CPC 6128

Zum besseren Verständnis sollten Sie sich wenigstens ganz grob mit der Verarbeitung der Bildschirmdaten Ihres CPC 6128 befassen. Eine intensive Abhandlung dieses Themas ist an dieser Stelle sicher nicht angebracht. Doch geben wir Ihnen hier wieder die Fakten, die Sie brauchen, um Turbo entsprechend einsetzen zu können.

- Der Bildschirmspeicher umfaßt 16 Kbyte. Er kann im Arbeitsspeicher an einer beliebigen Stelle plaziert werden, jedoch ist er unabhängig von der aktiven Bank immer in den ersten vier Speicherblöcken 0 bis 3 zu finden.

- Die Anfangsadresse dieses Video-RAM besteht zunächst aus einer Basisadresse SCR BASE, von der nur das Hi-Byte betrachtet und verwaltet wird.

  Beispiel: Wenn SCR BASE =$40, dann liegt die Basisadresse bei $4000.

- Der Bildschirm wird beim CPC nicht in Form von Matrizen, entsprechend denen des Zeichensatzes verwaltet, sondern ist in acht Rasterzeilen zerlegt, die jeweils 2 Kbyte belegen. Füllt man z.B. nur die erste Rasterzeile aus, dann erscheinen auf dem

Bildschirm 25 waagrechte, parallele Linien im Abstand von jeweils acht Bildschirmpunkten. Die zweite Rasterzeile liegt in gleicher Weise direkt darunter usw.

Wenn das erste Byte der ersten Rasterzeile z.B. bei $4000 liegt, findet man das erste Byte der zweiten Rasterzeile bei $6000 usw. Die letzten 48 Byte (2 Kbyte = 2048 Byte) sind jeweils unbenutzt, weil der Bildschirm selbst nur jeweils 2000 Byte darstellen kann.

Das bedeutet, daß z.B. ein Zeichen aus dem Zeichensatz aus acht Linien zusammengesetzt ist, wobei dazu 8 Byte des Bildschirm-RAM verändert werden müssen, um es auf dem Schirm auszugeben. Mit jedem Bit eines Bytes wird 1 Pixel (Bildschirmpunkt) gesetzt oder gelöscht. Dadurch ergibt sich auch die Bildschirmauflösung von 8*80 mal 25*8 = 640 mal 200 Pixeln.

- Mit einer zweiten Angabe, dem sogenannten SCR OFFSET, läßt sich außerdem noch einstellen, welche Bytenummer als erste am Bildschirmanfang stehen soll. SCR OFFSET wird dabei immer nur bis maximal $1FFF verwaltet, um nicht in die nächste Rasterzeile zu stoßen. Dieser Versatz von SCR OFFSET gilt aber für alle 8 Rasterzeilen.

  Beispiel: SCR BASE = $4000
  SCR OFFSET = $0200
  ergibt: Schirmanfang $4200
  Bedeutung: Die erste Bildschirmadresse wird mit dem Byte dargestellt, das in der Speicherstelle $4200 liegt.

- Die Speicheradressen $B7C4 und $B7C6 enthalten (außer in Bank 2, die von Turbo benützt wird) in allen Bänken den jeweils aktuellen Bildschirmanfang.

- Mit der Betriebssystem-Routine SET SCR OFFSET, deren Einsprung bei $BC05 liegt (außer bei Bank 2) kann ein neuer Offset eingestellt werden, der im Registerpaar HL übergeben werden muß.

- Mit der Routine SET SCR BASE (Einsprung bei $BC08, außer von Bank 0 aus) wird eine neue Basisadresse geschaltet, die im A-Register bereitgestellt sein muß.

- Auch Turbo bedient sich dieser Routinen und schaltet relativ häufig den Schirmanfang um.

Das mag vorläufig genügen, um die nächsten Programmteile verständlicher zu halten, die wir aus Platzgründen nun nicht mehr in aller Ausführlichkeit kommentieren werden.

17.9.2 Speichern einer Bildschirmseite auf Diskette Programm P88.SCR mit External »holblock«

Das Programm P88.SCR erfüllt folgende Aufgaben:

1. Laden der externen Routine »holblock«, die auf der Diskette unter dem Namen »SAVESCR.EXT« vorhanden sein muß, und Installieren dieses **Externals** in einem freien Speicherbereich (siehe dazu auch die Erläuterungen im Abschnitt 17.8.6). Dazu verwenden wir die bereits bewährte Prozedur »ExCodeIn« (siehe P81.DIR).

2. Einstellen des Schirmanfangs auf genau $4000, denn Turbo hat wahrscheinlich einen anderen Beginn gewählt.

3. Beschreiben des Schirms mit einer Zeichenfolge, die abgespeichert werden soll.

4. Die Prozedur »seitespeichern« holt sich mit Hilfe der **External**-Prozedur »holblock« in acht Schritten jeweils einen Block von 2000 Zeichen aus dem Bildschirm-RAM und speichert ihn mit der **BlockWrite**-Anweisung unter dem Namen »BildXX.GRA« auf Diskette.

```
      PROGRAM SchirmAufDisk; (* P88.SCR *)
      TYPE name=String[12];
      VAR i,anzahl,anfang:Integer;

      PROCEDURE holblock (anfang,anzahl:Integer);External $CE00;

(*A*) PROCEDURE schirmoffsetnull;
      BEGIN
        Inline($21/0/0/$cd/$5a/$fc/$05/$bc);
        Inline($3e/$40/$cd/$5a/$fc/$08/$bc);
      END;

      PROCEDURE seitespeichern;
      VAR
        i,zur,k:Integer;
        datei:File;
        aus:^Byte;
      BEGIN
```

```
(*B*)    aus:=Ptr($D000);
         Assign(datei,'BildXX.GRA');
         Rewrite(datei);
         FOR i:=0 TO 7 DO
         BEGIN
           holblock(2048*i,2000);
(*C*)      BlockWrite(datei,aus^,16,zur);
           GotoXY(1,18);write(zur);
         END;
         Close(datei);
       END;

       PROCEDURE ExCodeIn(datname:name);
       VAR
         datei:File;
         absfeld:Char Absolute $CE00;
       BEGIN
         Assign(datei,datname);
         Reset(datei);
         BlockRead(datei,absfeld,1);
         Close(datei);
       END;

       BEGIN
         ClrScr;
         FOR i:=1 TO 1000 DO Mem[$D000+i]:=0;
         ExCodeIn('SAVESCR.EXT');
         schirmoffsetnull;
         ClrScr;
         FOR i:=1 to 32 DO
         Write('^._=  MNOPQRSTUVWXYZ01234567890,|{///\\\|');
         seitespeichern;
         Write(#27,'p','fertig',#27,'q');
       END.
```

Erläuterungen:

(A) Die erste **Inline**-Routine setzt SCR OFFSET auf Null, die zweite setzt SCR BASE auf $4000. Beide Teile bedienen sich der Systemroutinen $BC05 bzw. $BC08 über den Einsprung $FC5A, den wir als äußerst bequemen Weg zum Betriebssystem schon kennengelernt haben (siehe Erläuterungen zu P77.GRA).

Wenn Sie diese **Inline**-Prozedur variabel halten wollen, dann zerlegen Sie sie z.B. in zwei einzelne Prozeduren: PROCEDURE scroffset (anfango:Integer); und PROCEDURE scrbase (anfangb:Integer); und ändern den Code in den

ersten **Inline**-Daten so ab, daß HL mit »anfango« bzw. A mit »anfangb« belegt wird.

(B) Der Puffer für die blockweise Übernahme der Bildschirmbytes wurde hier auf $D000 festgelegt, um ein Funktionieren des Demo-Programms zu garantieren. Zweckmäßigerweise legt man ihn zur praktischen Verwendung in einen geschützten Bereich, wie wir das oben beschrieben haben (17.8.6).

Aber Achtung: Auch die externe Routine »holblock« muß dann entsprechend angepaßt werden, kann aber mit einem weiteren Parameter ebenfalls variabel gehalten werden.

(C) Falls Sie die externe Prozedur »holblock« verbessern wollen, können Sie sich an diesem Assemblercode orientieren:

```
; ***************** »holblock« zu Programm P88.SCR ***************
;
   DI              ; Interrupts verhindern
   EXX             ; Registerinhalte retten
   POP HL          ; Rücksprungadresse holen
   POP DE          ; »anzahl« holen
   POP BC          ; »anfang« holen
   PUSH HL         ; Rücksprung ablegen
   EX (SP),HL      ; Stackpointer nach HL
   PUSH HL         ; und ablegen
   PUSH BC         ; »anfang« vorläufig ablegen
;  --------------- Ende der Stackvorbereitungen ----------------
   LD BC,#7FC0     ;
   LD A,193        ; Bank 1 wählen
   OUT (C),A       ; und einschalten
;  --------------- Ende Bankswitching 1 -------------------------
   LD HL,(#B7C4)   ; SCR OFFSET laden
   LD A, (#B7C6)   ; SCR BASE laden
   ADD A,H         ;
   LD H,A          ; und addieren
   POP BC          ; gewünschten »anfang« vom Stack holen
   ADD HL,BC       ; und addieren ==> SCR BEGIN
;  --------------- Ende Einstellung der Schirmanfangsadresse ----
   LD B,E
   LD C,D          ; Bytes aus »anzahl« vertauschen (Hi/Lo), um BC
                   ; als Zähler zu verwenden mit (B)=Lo
   LD DE,#CFFF ;Pufferanfang initialisieren(bei Bedarf ändern)
   DEC HL          ; Leseanfang initialisieren (-1)
   INC C           ; Zähler BC initialisieren
;  --------------- Ende der Initialisierungen -------------------
```

```
LOOP1:              ; Beginn der Außenschleife
   DEC C            ; Zähler Hi erniedrigen
LOOP2:              ; Beginn der Innenschleife
   INC HL           ; Leseadresse des Bildschirmspeichers erhöhen
   INC DE           ; Pufferadresse erhöhen
   LD A,(HL)        ; Bildschirmbyte laden
   PUSH BC          ; Zähler retten
   PUSH HL          ; Leseadresse retten
   LD H,A           ; Bildschirmbyte nach H retten
   LD A,194
   LD BC,#7FC0
   OUT (C),A        ; umschalten auf Bank 2
   LD A,H           ; Bildschirmbyte zur Ausgabe nach A
   LD (DE),A        ; Byte im Puffer ablegen
   LD A,193
   OUT (C),A        ; wieder auf Bank 1 zurückschalten
   POP HL           ; BS-Leseadresse vom Stack holen
   POP BC           ; Zähler vom Stack holen
   DJNZ LOOP2       ; Innenschleife wiederholen bis Zähler B=0
   LD A,0           ;
   ADD A,C          ; Hi-Byte des Zählers C auf Null prüfen
   JR NZ,LOOP1      ; Außenschleife anlaufen, bis auch C=0
;  --------------- Ende der Übertragung -------------------------
   LD A,194
   LD BC,#7FC0
   OUT (C),A        ; Bank 2 (Turbo-Bank) wieder einschalten
   POP HL
   EX (SP),HL       ; alten Stackpointer wieder einstellen
   EXX              ; alte Registerinhalte wieder herstellen
   EI               ; Interrupts zulassen
   RET              ; Rücksprung ins aufrufende Programm
; ***************** Ende der Prozedur »holblock« *****************
```

Anmerkung:

Wir haben diese Routine so aufgebaut, daß sie sowohl Daten zwischen verschiedenen Bänken übertragen, als auch durch kleine Änderungen andere Pufferbereiche ansprechen kann. In unserem Fall wäre ein Bankswitching nicht einmal erforderlich, weil der Puffer ab $D000 im Speicherblock 7 liegt, den die Banks 1 (CP/M) und 2 (Turbo) als gemeinsame Schnittstelle haben (siehe dazu 17.3). Wenn Sie jedoch eine COM-Datei aus dem Programm P88.SCR machen wollen, wo der Puffer unterhalb des Turbo-Codes liegt (siehe 17.8.6), dann ist dieses Verfahren unumgänglich.

17.9.3 Bildschirmseite von Diskette auf den Schirm Programm P89.SCR mit External-Prozedur »SchreibSchirm«

Das Programm P89.SCR ähnelt logischerweise stark dem Programm P88.SCR, weil es dessen Weg wieder zurückgeht:

1. **External**-Prozedur »SchreibSchirm« unter dem Namen FILLSCR.EXT von Diskette holen und in den Arbeitsspeicher einbauen.

2. Den Schirmanfang wieder auf $4000 einstellen, also in die Stellung bringen, bei der das Beschreiben und die Abspeicherung erfolgte.

3. In 2 Kbyte-Blöcken wird zunächst der Puffer ab $D000 gefüllt und dann wird von dort aus mit der externen Prozedur »schirmschreiben« der Pufferinhalt in den Bildschirmspeicher übertragen.

Wenn Sie dieses Programm ausprobieren, erkennen Sie anschaulich, wie die Erklärungen zum Bildschirmaufbau mit den Rasterzeilen vorhin zu verstehen waren.

```
PROGRAM Bildschirmseite_von_Diskette; (* P89.SCR *)
TYPE name=String[12];
VAR i,anzahl,anfang:Integer;
    taste:Char;

PROCEDURE SchreibSchirm(anfang,anzahl:Integer);External $CE00;

PROCEDURE schirmoffsetnull;
BEGIN
  Inline($21/0/0/$cd/$5a/$fc/$05/$bc);
  Inline($3e/$40/$cd/$5a/$fc/$08/$bc);
END;

PROCEDURE seiteladen;
  VAR
  i,k,zur:Integer;
  datei:File;
  zahl:Char;
  ein:Byte Absolute $D000;
BEGIN
  Assign(datei,'BILDXX.GRA');
  Reset(datei);
  FOR i:=0 TO 7 DO
  BEGIN
```

```
      BlockRead(datei,ein,16,zur);
      SchreibSchirm(i*2048,2000);
    END;
  END;

  PROCEDURE ExCodeIn(datname:name);
  VAR
    datei:File;
    absfeld:Char Absolute $CE00;
  BEGIN
    Assign(datei,datname);
    Reset(datei);
    BlockRead(datei,absfeld,1);
    Close(datei);
  END;

  BEGIN
    ClrScr;
    FOR i:=1 TO 1000 DO Mem[$D000+i]:=0;
    ExCodeIn('FILLSCR.EXT');
    ClrScr;
    schirmoffsetnull;
    seiteladen;
    Write(#27,'p','fertig',#27,'q');
  END.
```

Das **External** »SchreibSchirm« hat mit dem **External** »holblock« vieles gemeinsam:

Bis zum Einstieg in LOOP2 sind beide Routinen identisch. Kurz darauf unterscheiden sie sich wie folgt:

```
; *************** »SchreibSchirm« zu Programm P89.SCR *********
; ....
LOOP2:
   INC HL
   INC DE
   PUSH BC        ; Zähler und
   PUSH HL        ; Schirmzeiger retten
   LD BC,#7FC0    ;
   LD A,194
   OUT (C),A      ; Turbo-Bank 2 einschalten
   LD A,(DE)      ; und aus dem Puffer ein Byte holen
   LD H,A         ; Byte retten, weil A gebraucht wird
   LD A,193
   OUT (C),A      ; zum Umschalten auf Bank 1 (Video-RAM)
```

```
   LD A,H          ; A wieder mit Byte laden
   POP HL          ; Schirmadresse vom Stack holen
   LD (HL),A       ; und Byte dort ablegen
   POP BC          ; Zähler wieder vom Stack holen
   DJNZ LOOP2      ; und Innenschleife wiederholen, falls B<>0
   LD A,0
   ADD A,C         ; Zähler C schon auf Null?
   JR NZ,LOOP1     ; nein ===> Außenschleife wiederholen
   LD A,194        ; ja   ===> Turbo-Bank einschalten
   LD BC,#7FC0     ;
   OUT (C),A       ;
   POP HL          ; Stackpointer holen
   EX (SP),HL      ; und setzen
   EXX             ; Registerinhalte wieder holen
   EI              ; Interrupts zulassen
   RET             ; und zurück zur aufrufenden Routine
; ***************************************************************
```

Können Sie sich vorstellen,...

... daß man nun mit diesen Hilfen ein Programm erstellen kann, mit dem sich eine Bildschirmseite, z.B. ein Titelblatt oder eine Grafikseite von einem Spiel erstellen, abspeichern, bei nächster Gelegenheit wieder laden, verbessern und als Teil eines anderen Programms einsetzen läßt?

... daß Sie mit den Farben spielen und mit Hilfe der Schirmanfangsadressen oder der Betriebsroutinen den Schirm scrollen können?

... daß Sie durch Verstellen der Anfangsadressen oder durch Setzen von Hintergrund- und Zeichenfarbe ein Bild verdeckt aufbauen oder in einem anderen Speicher bereithalten können?

Dann erkennen Sie, welche Möglichkeiten Ihnen Turbo mit Hilfe ein paar weniger Kenntnisse aus der Maschinensprache und aus dem Betriebssystem zu bieten hat.

17.9.4 External-Programme direkt aus dem Arbeitsspeicher

Damit auch diejenigen in den Genuß unserer External-Routinen kommen, die (noch) nicht über ausreichende Assembler-Kenntnisse verfügen, bauen wir die beiden zuletzt besprochenen Programme so um, daß das Maschinenprogramm gleich mit in den Turbo-Quelltext integriert ist.

Das hat unter anderem den Vorteil, daß diese Programme auch lauffähig sind, ohne daß die **Externals** dafür auf Diskette bereitgehalten werden müssen. Wie Sie gleich sehen werden, nähert sich unsere Programmierweise etwas jener Art an, die wir bei der Besprechung der **Inline**-Anweisung kennengelernt haben.

Wir geben aber diesmal den Maschinencode in einem Byte-Array als Konstante vor und lesen ihn Byte für Byte in einen freien Speicherraum ein, so daß er von einer External-Prozedur aus aufgerufen werden kann.

Auf diese Weise können wir auch auf die Prozedur »ExCodeIn« verzichten, die uns bisher die auf Diskette ausgelagerten Maschinenprogramme hereingelesen hat. Es bleibt Ihnen diesmal also die Erstellung des Maschinencodes mit Hilfe eines Assemblers erspart. Das haben wir für Sie erledigt.

Vorsichtshalber – und weil Sie auf die Begleitdiskette verzichten müssen – drucken wir die Listings der beiden Bildschirm-Programme noch einmal vollständig ab. Wir haben sie »SaveBild« und »LiesBild« getauft.

```
      PROGRAM SaveBild;   (* P88PLUS.SCR *)
      TYPE name=String[12];
      VAR i,anzahl,anfang:Integer;

(*A*) PROCEDURE SaveExtIn;
      VAR i:Integer;
      CONST SaveExt: Array[0..81] Of Byte=
        (243,217,225,209,193,229,227,229,197,1,192,127,62,193,
         237,121,42,196,183,58,198,183,103,46,0,193,120,132,103,
         121,133,111,67,74,17,$FF,$CF,121,18,43,12,13,35,19,126,
         197,229,103,1,192,127,62,194,237,121,124,18,62,193,237,
         121,225,193,16,233,62,0,129,32,227,62,194,1,192,127,237,
         121,225,227,217,251,201);
      BEGIN
        FOR i:=0 TO 81 DO
        Mem[$CE00+i]:=SaveExt[i];
      END;

(*B*) PROCEDURE holblock (anfang,anzahl:Integer);External $CE00;

      PROCEDURE schirmoffsetnull;
      BEGIN
        Inline($21/0/0/$cd/$5a/$fc/$05/$bc);
        Inline($3e/$40/$cd/$5a/$fc/$08/$bc);
      END;
```

```
        PROCEDURE seitespeichern;
        VAR
          i,zur,k:Integer;
          datei:File;
          aus:^Byte;
        BEGIN
          aus:=Ptr($D000);
          Assign(datei,'BildXX.GRA');
          Rewrite(datei);
          FOR i:=0 TO 7 DO
          BEGIN
            holblock(2048*i,2000);
            BlockWrite(datei,aus^,16,zur);
            GotoXY(1,18);write(zur);
          END;
          Close(datei);
        END;

        BEGIN
          ClrScr;
          FOR i:=1 TO 1000 DO Mem[$D000+i]:=0;
(*C*)     SaveExtIn;
          schirmoffsetnull;
          ClrScr;
          FOR i:=1 to 32 DO
          Write('^._=  MNOPQRSTUVWXYZ01234567890,|{///\\\|');
          seitespeichern;
          Write(#27,'p','fertig',#27,'q');
        END.
```

Erläuterungen:

(A) Die Prozedur »SaveExtIn« schreibt die Codewerte des Maschinenprogramms in den Arbeitsspeicher, beginnend mit der Adresse $CE00. Die gesamte Routine besteht also aus 82 Zahlen. Sie sind aus dem Assemblerprogramm entstanden, das wir Ihnen unter dem Namen »holblock« in Abschnitt 17.9.2 vorgestellt haben.

(B) Am Aufruf des Externals ändert sich nichts.

(C) Im Hauptprogramm muß zur Vorbereitung das Maschinenprogramm installiert werden.

Nun das Gegenstück dazu, das ein abgespeichertes Bild wieder von der Diskette holt:

```
      PROGRAM LiesBild; (* P89PLUS.SCR *)
      TYPE name=String[12];
      VAR i,anzahl,anfang:Integer;
          taste:Char;

(*A*) PROCEDURE FillExtIn;
      VAR i:Integer;
      CONST FillExt: Array[0..80] Of Byte=
        (243,217,225,209,193,229,227,229,197,1,192,127,62,193,
         237,121,42,196,183,58,198,183,132,103,193,120,132,103,
         121,133,111,67,74,17,$FF,$CF,121,18,12,43,13,35,19,197,
         229,1,192,127,62,194,237,121,26,103,62,193,237,121,124,
         225,119,193,16,233,62,0,129,32,227,62,194,1,192,127,237,
         121,225,227,217,251,201);
      BEGIN
        FOR i:=0 TO 81 DO
        Mem[$CE00+i]:=FillExt[i];
      END;

      PROCEDURE SchreibSchirm(anfang,anzahl:Integer);External $CE00;

      PROCEDURE schirmoffsetnull;
      BEGIN
        Inline($21/0/0/$cd/$5a/$fc/$05/$bc);
        Inline($3e/$40/$cd/$5a/$fc/$08/$bc);
      END;

      PROCEDURE seiteladen;
        VAR
        i,k,zur:Integer;
        datei:File;
        zahl:Char;
       ein:Byte Absolute $D000;
     BEGIN
        Assign(datei,'BILDXX.GRA');
        Reset(datei);
        FOR i:=0 TO 7 DO
        BEGIN
          BlockRead(datei,ein,16,zur);
          SchreibSchirm(i*2048,2000);
        END;
      END;

      BEGIN
        ClrScr;
```

```
  FOR i:=1 TO 1000 DO Mem[$D000+i]:=0;
  FillExtIn;
  ClrScr;
  schirmoffsetnull;
  seiteladen;
  Write(#27,'p','fertig',#27,'q');
END.
```

Erläuterungen:

(A) Der Maschinencode gehört zu dem Programm, das wir aus dem Assembler-Quelltext »SchreibSchirm« (siehe 17.9.3) entwickelt haben.

Hinweise:

Wenn Sie die Lage des Puffers ändern wollen, in dem das Maschinenprogramm die Bildschirmdaten zwischenspeichert, dann ändern Sie die beiden Werte, die als einzige in den Konstanten-Arrays in Hexzahlen angegeben sind. Dort wird der Pufferanfang um eins erniedrigt mit $CFFF initialisiert.

Entsprechend müssen Sie die Anfangsadressen der Puffervariablen »aus« bzw. »ein« setzen.

Beispiel: Wenn Ihr Puffer bei $C000 beginnen soll, dann ändern Sie die Adresse in den Konstanten-Arrays (Lo/Hi) auf $FF/$BF und geben dann die Anweisungen

```
aus:=Ptr($C000);
```

bzw.

```
ein:Byte Absolute $C000;
```

Passen Sie aber auf, daß Sie nicht mit Ihren Bildschirmdaten Teile des Turbo-Programms, des **Externals** oder andere bereits belegte Bereiche überschreiben. In unserem Beispiel müßten bereits die externen Maschinenprogrammteile verlegt werden.

Was hier nicht getan wurde, können Sie bei Bedarf selbst einfügen:

- Abfrage, unter welchem Dateinamen das Bild gespeichert werden soll. (Hier wurde zur Demonstration die feste Datei »BILDXX.GRA« gewählt.)

- Absicherung gegen fehlerhafte Dateinamen. (Siehe dazu Kapitel 15!)

- Attraktive Bildschirmgestaltung.

17.10 Overlay-Technik spart Speicherplatz

Unter Overlay versteht man Überlagerungen von Programmteilen. Dabei wird für beliebig viele Unterprogramme der gleiche Bereich des Arbeitsspeichers benützt. Das Original-Turbo-Handbuch gibt darüber ausnahmsweise ausführlich Auskunft, so daß wir nur die wichtigsten Punkte an Hand eines Beispiels erläutern werden.

17.10.1 Das Prinzip der Overlay-Technik

- Eine Anzahl von Unterprogrammen (Prozeduren, Funktionen) wird in einem Overlay-Block auf Diskette abgelegt, wenn sie mit dem Zusatz **OVERLAY** versehen werden. Diese Arbeit übernimmt automatisch der Compiler. Auf Diskette entsteht eine **OVERLAY**-Datei, die mit dem Namen des Hauptprogramms und einer Erweiterung versehen wird, die die Nummer der Datei enthält.

- Im Hauptprogramm wird Speicherplatz für diese Unterprogramme reserviert. Und zwar soviel, daß das größte der Unterprogramme darin Platz findet.

- Beim Aufruf einer **OVERLAY**-Prozedur oder -Funktion, wird nur die eine aufgerufene Routine von Diskette in den freigehaltenen Speicherplatz geladen und ausgeführt.

- Auf diese Weise können beliebig viele Unterprogramme in ein Programm aufgenommen werden, die alle denselben Bereich im Arbeitsspeicher benützen.

- Damit wird aber gleichzeitig ausgeschlossen, daß ein Unterprogramm ein anderes desselben Overlay-Blocks aufrufen kann.

- Außerdem wird durch das Hereinholen des Unterprogramms von Diskette die Ausführungszeit länger, so daß diese Technik sich nicht für Programmteile eignet, die möglichst rasch ablaufen sollen.

17.10.2 Programmbeispiel P90.DEM: OVERLAY,$I

Wir setzen uns nun zum Ziel, zwei Programme aus dem vorigen Abschnitt in Overlay-Technik zum Laufen zu bringen, so daß wir eine Untersuchung der Diskette mit der Ausgabe des freien Speicherplatzes (Programm P79.DEM) und das Inhaltsverzeichnis (Programm P81) auf dem Bildschirm darstellen können.

Dazu gehen Sie bitte folgendermaßen vor:

- Laden Sie das Programm P79.DEM unter dem Namen »FREE.INC« als Workfile.
- Bauen Sie den Text von P79.DEM zu einer Prozedur um, indem Sie zu Beginn den Kopf »PROGRAM BDOS_Demo ...« durch »**PROCEDURE** speicherplatz und am Schluß den Punkt hinter dem **END** durch einen Strichpunkt ersetzen.
- Speichern Sie nun mit Turbo diesen Text unter »FREE.INC« ab.
- Das gleiche Spiel treiben Sie nun mit dem Programm P81.DIR, welches das Inhaltsverzeichnis ausgeben kann. Geben Sie ihm den Kopf **PROCEDURE** directory_lesen und legen Sie es unter dem Namen »DIRTORY.INC« ab.
- Wir besitzen nun zwei Prozeduren, die wir als **Include**-Dateien verwenden können.

Das Hauptprogramm wird nun so einfach, wie ein Turbo-Programm nur sein kann:

```
(*A*) PROGRAM OVRLAY; (* P90 *)
(*B*) OVERLAY {$I DIRTORY.INC}
(*C*) OVERLAY {$I FREE.INC}
(*D*) BEGIN
        directory_lesen;
        speicherplatz;
      END.
```

Erläuterungen:

(A) Beim Programmnamen wurde das »E« absichtlich weggelassen, da das Wort **OVERLAY** als reserviertes Wort nicht verwendet werden darf.

(B) Nach **OVERLAY** steht nun die Prozedur, die an dieser Stelle eingebaut werden kann.

(C) Die zweite **OVERLAY**-Prozedur gehört zum selben Overlay-Block, weil keine andere Anweisung zwischen den beiden Prozeduren mehr steht.

(D) Das Hauptprogramm ruft zunächst »director_lesen« und anschließend »speicherplatz« auf. Die Reihenfolge der Aufrufe ist genauso unerheblich wie die Reihenfolge der **OVERLAY**-Deklarierungen.

Für die folgenden Schritte sind folgende Programme auf Ihrer Diskette erforderlich: TURBO.COM, TURBO.OVR, DIRTORY.INC, FREE.INC und die als **External** benötigte Prozedur SEKTOR.EXT.

Bevor Sie Programm P90 ausprobieren können, muß es kompiliert werden. Aber Achtung! Dieses Mal muß der Compiler eine COM-Datei erzeugen. Dazu gehen Sie ins Hauptmenü von Turbo und geben **<O>** für die Compiler-Optionen ein, dann **<C>** für die Erstellung von COM-Dateien und verlassen dieses Sub-Menü wieder mit **<Q>**. Jetzt tippen Sie wieder vom Hauptmenü aus **<C>**, und der Compiler erzeugt den Code: Sie hören und sehen, wie die Diskette anläuft und das I vor den Zeilennummern erscheint. Die Unterprogramme werden also zunächst als Turbo-Text von Diskette geholt, kompiliert und anschließend aber nicht in das Hauptprogramm installiert, wie wir das sonst von **Include**-Dateien kennen, sondern als **OVERLAY**-Datei auf Diskette ausgelagert.

Zum Ausprobieren haben Sie diesmal zwei Möglichkeiten:

– Verlassen Sie Turbo mit **<Q>** und tippen Sie »P90« wie jeden anderen CP/M-Befehl ein, dann wird P90.COM aufgerufen.

– Aus Turbo heraus dürfen Sie diesmal aber nicht mit **<R>** starten, sondern müssen **<X>** für Execute wählen. Diese Option ist extra für den Lauf von COM-Programmen eingerichtet.

Aber noch einmal Achtung! Wenn Sie **<X>** verwenden, muß auf Ihrer Diskette auch noch die Datei TURBO.OVR vorhanden sein, die Sie mit Ihrer Original-Diskette erhalten haben. Andernfalls bricht Turbo ab und beklagt sich, daß es ohne dieses Hilfsprogramm nicht weiterarbeiten kann.

Der Grund: Teile von Turbo (der Editor und der Compiler) werden beim Aufruf von **<X>** entfernt (intelligenterweise aber auch nur dann, wenn eine Datei namens TURBO.OVR auf Diskette vorgefunden wurde) und dafür TURBO.OVR und die gewünschte COM-Datei geladen. Nach Beendigung des Laufs des COM-Programmes sorgt TURBO.OVR dafür, daß Turbo wieder einsatzbereit installiert wird.

Gleichgültig für welche Art, das Programm zu starten, Sie sich entscheiden, können Sie folgendes registrieren:

1. Wieder läuft die Diskette an, diesmal aber, um sich aus der **OVERLAY**-Datei die Prozedur »directory_lesen« zu holen und sie in den freigelassenen **OVERLAY**-Bereich einzubauen.

   Das gleiche geschieht kurz darauf auch mit der Prozedur »speicherplatz". Sie überlagert (daher Overlay) »directory_lesen«, weil sie in den gleichen Raum gesteckt wird wie ihre Vorgängerin.

2. Wenn das Inhaltsverzeichnis ausgegeben wird, erscheint eine Datei namens »P90.000". Das ist die **OVERLAY**-Datei, die den Code der beiden Prozeduren enthält.

Selbstverständlich können Sie auch **OVERLAY**-Dateien erzeugen, indem Sie den Text Ihrer Prozeduren und Funktionen in den Text Ihres Hauptprogramms integrieren. Wir haben hier nur deshalb die Möglichkeit mit den **Include**-Dateien vorgestellt, weil das Handbuch darauf nicht eingeht. Denn erst mit dieser Variante beginnt die Platzersparnis und Ihre Turbo-Programme bekommen damit profihaftes Format.

- Jeder Programmteil wird streng für sich entwickelt und lauffähig gemacht.

- Notwendige Programmteile werden als Prozeduren oder Funktionen auf Diskette übertragen.

- Der Einbau in das Hauptprogramm kann nun mit einfachen **Include**-Anweisungen oder mit **OVERLAY** oder mit der eben vorgestellten Kombination aus beiden Möglichkeiten erfolgen.

17.10.3 Geteilte OVERLAY-Blöcke, Programm P91.DEM

Sie können Turbo aber auch dazu zwingen, mehrere **OVERLAY**-Dateien für ein und dasselbe Hauptprogramm anzulegen. Das verbraucht aber viel Speicherplatz, weil dann auch mehrere Bereiche reserviert werden.

Sollte dies aus irgendeinem Grund einmal sinnvoll sein, weil z.B. eine **OVERLAY**-Prozedur eine andere **OVERLAY**-Funktion benötigt, dann genügt es, im Turbo-Text die einzelnen Overlay-Blöcke durch dazwischenliegende andere Teile zu trennen.

Sie sollten das einmal ausprobieren, indem Sie unser Beispiel P78 wie folgt umbauen:

```
      PROGRAM Overlay_Getrennt;   (* P91.DEM *)
      OVERLAY {$I DIRTORY.INC}
(*A*) PROCEDURE dummy;
      BEGIN END;
      OVERLAY {$I FREE.INC}
      BEGIN
        speicherplatz;
        directory_lesen;
      END.
```

Wir haben bei (A) lediglich eine Scheinprozedur eingefügt, die eigentlich gar nichts bewirkt, was zum Programmablauf gehört.

Wenn Sie dieses Programm aber unter dem Namen P91.DEM wie oben beschrieben mit Option **<C>** kompilieren und sich hinterher das Inhaltsverzeichnis Ihrer Diskette anschauen, werden Sie feststellen, daß plötzlich außer der Datei P91.COM noch zwei **OVERLAY**-Dateien entstanden sind, nämlich »P91.000« und »P91.001«. Die erste enthält den Code für »directory_lesen«, die zweite den Code für »speicherplatz«.

Sinnvoll ist unser Testprogramm P91.DEM natürlich nicht aufgebaut, denn im ersten freien Overlay-Bereich ist auch Platz für die Prozedur selbst. Und da keine zweite mehr für diesen Bereich vorgesehen ist, kann dort auch gleich das Unterprogramm selbst stehen. Die Ausführungszeit verkürzt sich dann um die Zeit für das Laden von Diskette. Und das gleiche gilt auch für den zweiten freigehaltenen Speicherraum.

Zusammenfassung und Ergänzungen:

- **OVERLAY**-Dateien werden vom Compiler auf Diskette angelegt und fortlaufend numeriert, beginnend bei 000.

- Die Kompilierung darf nicht im Memory-Modus (Option **<M>**) erfolgen, sondern muß mit Option **<C>** oder **<H>** durchgeführt werden.

- Je mehr Unterprogramme in einer einzigen **OVERLAY**-Datei untergebracht sind, desto größer wird die Platzersparnis.

- Zur Ausführung wird jedes **OVERLAY**-Programm erst von Diskette in den Arbeitsspeicher geholt. Der Platzbedarf richtet sich hier nach dem größten Unterprogramm einer **OVERLAY**-Datei.

- **OVERLAY**- und **Include**-Technik lassen sich kombinieren.

- Geschachtelte **OVERLAYs** sind zulässig. Rekursive Unterprogramme sind in **OVERLAYs** nicht möglich.

- Normalerweise werden die durch **OVERLAY** aufgerufenen Unterprogramme immer im angemeldeten Laufwerk gesucht. Sie können aber auch dieses Laufwerk selbst bestimmen, indem Sie die Prozedur **OvrDrive** aufrufen und als Parameter das Laufwerk mitgeben.

  Beispiel: `OvrDrive(1)`

  Funktion: Alle **OVERLAYs** werden ab sofort in dem Laufwerk gesucht, das mit B bezeichnet wurde, was nur dann sinnvoll ist, wenn man entweder zwei Laufwerke angeschlossen hat oder einen Diskettenwechsel erzwingen will.

17.11 Verketten von Programmen

17.11.1 Execute und Chain

In sich abgeschlossene, kompilierte Programme können unter Turbo von Diskette abgerufen und gestartet werden. Dabei wird der Programmcode des ursprünglichen, aufrufenden Programms durch den neuen Code ersetzt.

Zwei Möglichkeiten stehen dabei zur Verfügung:

– **Execute** aktiviert ein Programm mit dem Zusatz .COM. Falls sie mit Turbo-Pascal erstellt wurden, sind sie lauffähig, da sie den notwendigen Routinenblock mitführen, der als Library bezeichnet wird.

– **Chain** lädt ein Turbo-Programm, das den Zusatz .CHN trägt, und startet es. Solche Programmteile enthalten nur den reinen Programmcode ohne die sogenannte Library. Sie sind damit erheblich kürzer, können aber nur laufen, wenn sich bereits die Library im Arbeitsspeicher befindet.

In welcher Form ein Programm abgespeichert wird, kann man im Turbo-Menü vorwählen, und zwar mit dem Tastendruck **<O>** für Optionen. Diesen Punkt haben wir bisher ausgespart. An einem Beispiel werden wir nun seine Arbeitsweise kennenlernen.

17.11.2 Verknüpfung von Programmen mit Execute Programm P92 »EXEPROG«: Assign, File, Execute

Wir setzen uns zum Ziel, zwei bereits vorhandene Programme von einem Hauptprogramm aus nacheinander zu laden und zu starten. Schematisch ergibt sich folgender Ablauf:

– Hauptprogramm ruft erstes Diskettenprogramm

– erstes Diskettenprogramm ruft Hauptprogramm

– Hauptprogramm ruft zweites Diskettenprogramm

– zweites Diskettenprogramm ruft Hauptprogramm

Ideal wird das Ganze aufgebaut, wenn das Hauptprogramm ein Menü erzeugt, das uns die Wahl des aufzurufenden Programms freistellt. Es ist aber auch ohne weiteres möglich, daß sich die Diskettenprogramme immer gegenseitig aufrufen.

Unser Hauptprogramm soll je nach Wahl eines der beiden Programme aufrufen, die wir auch im vorigen Abschnitt als Prozeduren verwendet haben:

– »EXEDISK.COM« untersucht die Diskette auf freien Speicherplatz.

– »EXEDIR.COM« gibt das Inhaltsverzeichnis aus.

Zunächst das Hauptprogramm P92 »EXEPROG.PAS«:

```
      PROGRAM ExeProg; (* EXEPROG.PAS *)
      VAR
(*A*)   programm:File;
        auswahl:Integer;
      BEGIN
        ClrScr;GotoXY(1,12);
(*B*)   Write('Beenden <0>   Diskuntersuchung  <1> ');
        Write('      Inhaltsverzeichnis    <2> ');
        REPEAT
          ReadLn(auswahl);
          CASE auswahl OF
(*C*)     1: BEGIN
               Assign(programm,'EXEDISK.COM');
               Execute(programm);
             END;
```

```
      2: BEGIN
           Assign(programm,'EXEDIR.COM');
           Execute(programm);
         END;
       END;
     UNTIL auswahl=0;
   END.
```

Erläuterungen:

(A) Die **Execute**-Programme behandeln wir als allgemeines **File** und nennen die Dateivariablen, mit deren Hilfe sie aufgerufen werden, einfach »programm«.

(B) Das Menü beschränkt sich auf drei Punkte. Ein Beenden sollte grundsätzlich immer vorgesehen werden.

(C) Um ein Programm mit **Execute** aufzurufen, muß es vorher mit **Assign** angesprochen werden. Die Vorbehandlung erfolgt also wie bei allen anderen Dateien.

Auch wenn wir schon die **Execute**-Programme, also COM-Programme, die ausführbaren Code beinhalten, auf Diskette hätten, können wir P91.DEM noch nicht ausprobieren, weil **Execute** (und auch **Chain**) nicht im Memory-Modus verwendet werden dürfen. Also kompilieren wir dieses Programm wieder mit der Option **<C>**.

Damit steht nun das Hauptprogramm unter dem Namen »EXEPROG.COM« auf Diskette bereit. Merken Sie sich bitte, daß es immer mit der Option **<C>** kompiliert werden muß. Nur so ist es in der Lage, als eigenständiges Programm aufgerufen werden zu können, das wiederum andere Programme aktivieren kann.

17.11.3 Vorbereitung der Execute-Programme

Wenn Sie unserem Beispiel folgen, und die beiden Diskettenprogramme einbauen wollen, dann holen Sie erst einmal das Programm P79.DEM unter dem Arbeitsnamen »EXEDISK« herein und fügen unter **VAR** folgendes ein:

```
programm: File;
```

Vor dem abschließenden **END.** brauchen wir dann den Aufruf für das Hauptprogramm, der uns zurück ins Menü bringt. Damit wir auch das Ergebnis betrachten können, fügen wir noch die bekannte Warteschleife mit **KeyPressed** ein:

```
REPEAT UNTIL KeyPressed;
Assign (programm,'EXEPROG.COM');
Execute(programm);
```

Speichern wir erst einmal wieder »EXEDISK« als Turbo-Text ab, falls noch weitere Änderungen oder Verbesserungen notwendig werden. Anschließend kompilieren wir mit der Option **<C>** diesen Text.

Das gleiche, was wir soeben mit Programm P79.DEM getan haben, unternehmen Sie nun bitte auch mit dem zweiten (z.B. Programm P81.DIR, das wir »EXEDIR« nennen wollten) und – wenn Sie eigene Programme verketten – mit allen weiteren Programmen, die Sie verwenden wollen.

Auf Diskette sollten Sie nun u.a. folgende Dateien finden:

```
EXEPROG.COM
EXEDISK.COM
EXEDIR.COM
```

Und jetzt wird es spannend. Wir starten zum Testlauf:

Schalten Sie nun Turbo ab, indem Sie mit **<Q>** aussteigen. Unter CP/M rufen Sie nun einfach »EXEPROG«. Dieses .COM-Programm hat alles, was es zum Laufen braucht und es beginnt folgerichtig mit dem Menü.

Wählen Sie nun einen Punkt aus, dann startet das Laufwerk und holt sich das gewünschte **Execute**-Programm. Dieses wird ausgeführt und anschließend wird wieder das Hauptprogramm geladen, sobald irgendein Tastendruck erfolgt ist.

17.11.4 Chain-Programme sparen Diskettenplatz

Wenn Sie sich den Platzbedarf der soeben erzeugten COM-Dateien im Inhaltsverzeichnis ansehen, dann stellen Sie fest, daß etliche Kbytes pro Datei verbraucht werden, obwohl die Programme selbst recht kurz ausgefallen sind.

Das kommt daher, daß jedes dieser Programme den sogenannten Runtime-Pack, die vorhin schon erwähnte Turbo-Library, mit sich führt.

Nun ist es aber nicht notwendig, daß jeder nachzuladende Teil für sich lauffähig ist. Es genügt, wenn das Hauptprogramm mit dem Runtime-Pack versehen ist.

Und so ein abgespecktes Programm, das nur aus dem Programmcode besteht, läßt sich mit der Compiler-Option **<H>** erzeugen.

Sie tun dazu folgendes:

1. Laden des Turbo-Textes (z.B. »EXEDISK.PAS«) als Arbeitsdatei und Abändern des Namens für das Menüprogramm in »CHNPROG.COM«,

2. aus dem Turbo-Menü die Compiler-Optionen mit **<O>** anlaufen,

3. mit **<H>** die Chain-Version wählen,

4. mit **<Q>** aus dem Optionenmenü aussteigen,

5. mit **<C>** kompilieren.

Auf Diskette finden Sie nun die neue Datei EXEDISK.CHN. Mit allen weiteren Dateien gehen Sie genauso vor.

Das Hauptprogramm »EXEPROG.PAS« muß jetzt nur in zwei Kleinigkeiten im Abschnitt (C) abgeändert werden.

6. Ersetzen Sie in den Dateinamen einfach COM durch CHN:

```
... Assign (programm,'EXEDISK.CHN'); ...
... Assign (programm,'EXEDIR.CHN'); ...
```

7. Ersetzen Sie **Execute** durch **Chain**.

Diese Änderungen nehmen Sie am besten so vor, daß Sie eine neue Arbeitsdatei »CHNPROG« anfordern, die natürlich als »New File« keinen Text enthält. Aber mit Hilfe des Editors können Sie über **<CTRL/K>** und **<R>** den bereits vorhandenen Text von »EXEPROG« einlesen und ihn nach Bedarf abändern.

Das neue Hauptprogramm »CHNPROG.PAS« wird sicherheitshalber erst einmal abgelegt und dann wie oben beschrieben zur COM-Datei kompiliert, wobei wir uns vergewissern sollten, daß die Startadresse, die im Optionen-Menü veränderbar ist, die gleiche ist wie beim Kompilieren der Chain-Programme.

Auf Diskette sollten Sie nun unter anderem folgende Dateien finden:

```
CHNPROG.COM
EXEDISK.CHN
EXEDIR.CHN
```

Wenn Sie sich den Speicherbedarf auf der Diskette ansehen, dann stellen Sie fest, daß die CHN-Dateien wesentlich genügsamer mit dem Platz umgehen. Der Testlauf erfolgt wieder durch einfaches Aufrufen von CHNPROG von CP/M aus oder durch **<X>** von Turbo aus. Dabei fällt auf, daß die CHN-Programme schneller aktiv sind, weil nur der Programmcode ohne Library geladen werden muß.

Sie sehen, auf diese Weise kann man Programme schreiben, deren Umfang nur noch von der Kapazität der Disketten begrenzt wird. Und auch das läßt sich noch erweitern, wenn man den Anwender die Diskette wechseln läßt.

Zusammenfassung und Ergänzungen:

- Mit **Execute** werden Programme geladen und gestartet, die nicht auf die Turbo-Library angewiesen sind.

- Mit **Chain** werden Programme geladen und gestartet, die auf die Turbo-Library angewiesen sind.

- Programme, die **Execute**- oder **Chain**-Anweisungen enthalten, sind nicht lauffähig, wenn sie mit der voreingestellten Option **<M>** kompiliert werden.

- Befindet sich bereits ein Programm im Speicher, das als COM-Datei (Option **<C>**) erzeugt wurde, dann wird beim Programmlauf Zeit gespart, wenn weitere Disketten-programme CHN-Programme sind.

Übrigens: Wenn Sie Variablen aus dem vorhergehenden Programm in das nächste übernehmen wollen, dann deklarieren Sie diese in beiden Programmen zu Beginn des **VAR**-Teils in der gleichen Reihenfolge als globale oder absolute Variablen. Wir haben schon früher festgestellt, daß Turbo so schnell keine Variableninhalte vergißt. Das läßt sich hier vorteilhaft ausnützen, da bei einem neuen Programmstart die Variablen nicht automatisch zurückgesetzt werden, wie das z.B. bei den meisten BASIC-Versionen der Fall ist.

Anhang

A

FileHandling – ein nützliches Dienstprogramm

Hier nun das versprochene Dienstprogramm, das Ihnen bei der Bearbeitung Ihrer Diskettendateien bald unentbehrlich sein wird. Bauen Sie es nach Belieben weiter aus:

```
(*************************************************************)
(*          TURBO PASCAL auf dem Schneider CPC 6128          *)
(*              Behandlung von Diskettendateien              *)
(*                 Markt & Technik MT 90455                  *)
(*                   (C) W. Kassera 1987                     *)
(*************************************************************)

PROGRAM FileHandling;
TYPE
  L12=String[12];
VAR
  Quelle,Ziel:FILE;
  QDatei,ZDatei,DatName: String[12];
  taste,DatFlag:Integer;
```

```
PROCEDURE SektorExtIn;
VAR i:Integer;
CONST
  SektorExt: ARRAY[0..46] OF Byte=
  (243,217,225,193,209,229,227,229,197,213,62,193,1,192,127,237,
  121,1,0,208,205,36,252,193,205,30,252,193,205,33,252,205,
  39,252,62,194,1,192,127,237,121,225,227,217,251,201,201);
BEGIN
  FOR i:=0 TO 46 DO
  MEM[$C000+i]:=SektorExt[i];
END;

PROCEDURE Datei_Name(VAR DatName:L12);
VAR
  ch:Char;
BEGIN
  REPEAT
    Write(^j^m,#27,'A','Dateiname: '); ClrEol;
    Read(DatName);
    Assign(Quelle,DatName);
    {$I-}
    Reset(Quelle);
    {$I+}
    IF IOResult=0 THEN
    BEGIN
      Write(' existiert. ');
      DatFlag:=1;
    END;
    ELSE
    BEGIN
      Write(' ist neu. ');
      DatFlag:=0;
    END;
    Write(' Korrekt?  J/N');
    Read(KBD,ch);
  UNTIL UpCase(ch)='J';
    Close(Quelle);
END;

PROCEDURE Directory_Lesen;   (* P81.DIR *)
TYPE
  Name=String[12];
VAR
  sektor,device: Integer;

PROCEDURE LiesSektor (spur,sektor: Integer); EXTERNAL $C000;

PROCEDURE LiesPuffer;
```

```
TYPE
  L11=Array[0..11] OF Char;
  ZeigerTyp=^L11;
VAR
  i,j,k,n: Integer;
  Zeiger: Zeigertyp;
BEGIN
  {Pufferanfang $D000 ist im ext. Maschinenprogramm festgelegt}
  k:=$D000-32; n:=0;
  FOR i:= 0 to 15 DO
  BEGIN
    k:=k+32;
    Zeiger:=PTR(k);
    IF((ORD(Zeiger^[0]) AND 128 =0) AND (Zeiger^[1]<>#255)) THEN
    BEGIN
      FOR j:= 1 to 11 DO
      Write(USR,Zeiger^[j]);
      Write(USR,'     ');
      n:=n+1;
      IF n=4 THEN
      BEGIN
        n:=0; WriteLn(USR);
      END;
    END;
  END;
  WriteLn(USR);
END;

PROCEDURE Vorbereitung;
VAR i:Integer;
BEGIN
  ClrScr;
  Write ('Inhaltsverzeichnis auf Schirm <0>  Drucker <1> : ');
  ReadLn(device);
  IF device=1 THEN
  BEGIN
    Write(LST,#27,'l',chr(10));
    USROUTPTR:=LSTOUTPTR;
  END
  ELSE USROUTPTR:=CONOUTPTR;
  FOR i:=1 TO 512 DO MEM[$D000+i]:=0;
  IF MEM[$C010]<>121 THEN
  SektorExtIn;
END;
BEGIN
  Vorbereitung;
  FOR sektor:=0 to 3 DO
```

```
  BEGIN
    LiesSektor(2,sektor);
    LiesPuffer;
  END;
  IF taste=4 then
  BEGIN
    GotoXY(1,24);Write('Taste druecken!');
    REPEAT UNTIL Keypressed;
  END;
END;

PROCEDURE FILECOPY;
CONST
  Pufferzahl=100;
  Recordzahl=128;
VAR
  Quelle,Ziel:FILE;
  QDatei,ZDatei: String[12];
  Puffer:Array[1..128,1..100] OF Byte;
  Zeiger,kopiert,gelesen,DatFlag:Integer;
BEGIN
  ClrScr;Directory_Lesen;
  WriteLn(#27,'p','Copy-Programm fuer beliebige Dateien',#27,'q');
  Write(^j^m,'Kopiere von: '^j^m);
  Datei_Name(QDatei);
  IF DatFlag <>0 THEN
  BEGIN
    Write(^j^m,'auf Datei: ');
    ReadLn(ZDatei);
    Assign(Quelle,QDatei); Assign(Ziel,ZDatei);
    Zeiger:=0;
    REPEAT
      WriteLn('Quelldiskette einlegen, dann Taste druecken!');
      REPEAT UNTIL KeyPressed;
      Reset(Quelle);
      SEEK(Quelle,Zeiger);
      BlockRead(Quelle,Puffer,80,gelesen);
      WriteLn(gelesen,' Blocks gelesen.');
      Close(Quelle);
      WriteLn('Zieldiskette einlegen, dann Taste druecken!');
      REPEAT UNTIL KeyPressed;
      IF zeiger=0 THEN ReWrite(Ziel) ELSE Reset(Ziel);
      SEEK(Ziel,Zeiger);
      BlockWrite(Ziel,Puffer,gelesen,kopiert);
      WriteLn(kopiert,' Blocks kopiert.');
      Close(Ziel);
      Zeiger:=Zeiger+gelesen;
    UNTIL gelesen<80;
```

```
    Write('Kopieren beendet.'); Delay(1000);
  END;
END;

PROCEDURE NameNeu;
VAR NeuName:L12;
BEGIN
  ClrScr; Directory_Lesen;
  WriteLn(#27'p',^J^M,'Umbenennen einer Datei',#27,'q',^j^m);
  Datei_Name(QDatei);
  IF DatFlag <>0 THEN
  BEGIN
    WriteLn(^j^m,'auf: ');
    REPEAT
      Write(^j^m,#27,'A');Datei_Name(NeuName);
    UNTIL DatFlag=0;
    Assign(Quelle,Qdatei);
    Rename(Quelle,NeuName);
    Close(Quelle);
  END;
END;

PROCEDURE Scratch;
BEGIN
  ClrScr; Directory_Lesen;
  WriteLn(#27,'p',^j^m'Loeschen einer Datei',#27,'q');
  Datei_Name(DatName);
  IF DatFlag=1 THEN
  BEGIN
    Assign(Quelle,DatName);
    Erase(Quelle);
    Close(Quelle);
  END;
END;

BEGIN
  REPEAT
    ClrScr;
    WriteLn(#27,'p',#27,'2','2',' Programm zur Datei-Behandlung ');
    Write('      (C) W. Kassera 1987        ',#27,'q',^j^j^m);
    WriteLn('   Kopieren            <1>');
    WriteLn('   Loeschen            <2>');
    WriteLn('   Umbenennen          <3>');
    WriteLn('   Inhaltsverzeichnis  <4>');
    WriteLn(^j^j^m,'   Beenden              <0>');
    Write(^j^j^m,'   Auswahl Taste:       ');
    BufLen:=1; Read(taste);
    CASE taste OF
```

```
        1: FileCopy;
        2: Scratch;
        3: NameNeu;
        4: Directory_Lesen;
    END;
  UNTIL taste = 0;
  ClrScr;
  Write('Programm beendet.'#27,'2','0');
END.
```

Anhang

B

Prozeduren und Funktionen in Turbo-Pascal

Da es zeitraubend ist, im Handbuch ständig nach irgendwelchen Bezeichnern und ihrer Syntax blättern zu müssen, wenn man sich der Anwendung bzw. Anwendbarkeit nicht sicher ist, stellen wir im folgenden eine alphabetische Liste aller Turbo-eigenen Funktionen und Prozeduren zusammen. Prozeduren wurden mit einem »P« gekennzeichnet, Funktionen mit einem »F«. Wir unterscheiden das deshalb, weil bei Funktionen nach ihrer Ausführung der Inhalt sofort unter dem Funktionsnamen greifbar und weiterverarbeitbar ist. Bei Prozeduren ist das nicht der Fall.

Achtung! Nicht alle aufgeführten Anweisungen sind in anderen Pascal-Dialekten verwendbar. Turbo hat eine ganze Reihe eigener Kommandos, die das Programmieren erleichtern.

Die Standardbezeichner stehen am linken Rand. Das sind die Wörter, die Sie am besten nie für Ihre eigenen Bezeichnungen von Variablen, Konstanten, Funktionen usw. verwenden sollten. In einem String dagegen dürfen sie selbstverständlich vorkommen. Rechts daneben finden Sie die Schreibweise in allgemeiner Form. Darunter ein Beispiel, das nur den Anweisungsteil eines Programms enthält.

Den Deklarationsteil, **BEGIN**, **END** usw. haben wir weggelassen, weil es sich hierbei fast immer um das gleiche Schema handelt.

Bei Funktionen oder Prozeduren, die keine Parameter verlangen, verzichten wir auf ein Beispiel, weil sie einfach mit ihrem Namen aufgerufen werden.

Addr
(F)

`Addr(Bezeichner)`
holt die Speicheradresse des ersten Bytes, wo eine Variable, Funktion oder Prozedur mit dem Namen »Bezeichner« beginnt.

Beispiel:
```
anfang_gradin:=Addr(gradin);
Write(anfang_gradin);
```
Ergebnis: eine ganze Zahl aus dem Adreßbereich

ArcTan
(F)

`ArcTan(tangens)`
berechnet den Winkel von einem Tangenswert im Bogenmaß

Beispiel:
```
winkel:=ArcTan(1.0);
Write(winkel*180/pi:4:1);
```
Ergebnis: »45.0«

Assign
(P)

`Assign(dateivariable, name)`
weist »dateivariable« einen Namen (String) zu.

Beispiel:
```
Assign(file,"orte«);
Reset(file);
```
Ergebnis: Datei »orte« ist zum Bearbeiten offen.

BlockRead
(P)

`BlockRead(file, variable, anzahl, <kontrolle>);`
liest eine »anzahl« Blöcke (jeweils 128 Byte) über eine »variable« aus der Datei »file«. Voraussetzungen: geöffnete Datei »file« »variable« muß alle Bytes von BlockRead fassen.

Beispiel:
```
BlockRead(orte,west,2,k);
Write(west);
```
Ergebnis: max. die ersten 256 Zeichen der Datei »orte«

BlockWrite
(P)

`BlockWrite(file,variable,anzahl,<kontrolle>);`
schreibt entsprechend eine »anzahl« Datensätze zu je 128 Byte über eine »variable« in die geöffnete Datei »file«.

Beispiel:
```
BlockWrite(orte,west,1,k);
```
Ergebnis: Übertragung von max. 128 Byte über die Variable »west« an den Anfang der Datei »orte«. Die Variable »kontrolle« enthält die Zahl der tatsächlich übertragenen Datensätze.

Chain `Chain(chnfile)`
(P) lädt und startet ein mit **<H>** kompiliertes Programm. Zuvor ist Assign auszugeben.
Beispiel:
```
Assign(chnfile,'inhalt.chn');
Chain(chnfile);
```
Ergebnis: Das bereits compilierte Programm INHALT.CHN wird von Diskette als Hauptprogramm übernommen.

Chr `Chr(Nummer)`
(F) gibt das unter »Nummer« geordnete Zeichen aus dem Zeichensatz aus.
Beispiel: `Write(Chr(87));`
Ergebnis: »W«

Close `Close(file)`
(P) schließt die Datei unter der Dateivariablen »file«.
Beispiel: `Close(orte);`

ClrEoL `ClrEoL`
(P) löscht alle Zeichen auf dem Schirm, die rechts vom Cursor stehen.

ClrScr `ClrScr`
(P) löscht den Bildschirm und setzt den Cursor links oben in die Ecke.

Concat `Concat(string1,string2,...stringN)`
(F) verknüpft Strings bis zu einer Länge von 255 Zeichen.
Beispiel:
```
string1:='Pascal';
string2:='Turbo';
string3:='-';
Write(Concat(string2,string3,string1);
```
Ergebnis: »Turbo-Pascal«

Copy `Copy(string,von,anzahl)`
(F) bildet einen »anzahl« langen Teilstring aus einem »string« mit Zeichen Nummer »von« beginnend.
Beispiel:
```
teil:= Copy('Handbuch',5,4);
Write(teil);
```
Ergebnis: »buch«

Cos
(F)

`Cos(winkel)`
berechnet den Cosinuswert von einem »winkel«, der im Bogenmaß gegeben ist. Das Ergebnis liegt im Bereich von -1 bis +1.
Beispiel: `Write(Cos(pi/3):1:2));`
Ergebnis: 0.50

CtrExit
(P)

`CtrExit`
sendet den sogenannten Terminal-Reset-String zum Bildschirm.
Beispiel: siehe Handbuch »Installierung«

CtrInit
(P)

`CtrInit`
sendet den sogenannten Terminal-Initialization-String zum Bildschirm. Siehe Handbuch.

DelLine
(P)

`DelLine`
löscht die Zeile, in der der Cursor gerade steht. Alle darunterliegenden Zeilen werden nachgeschoben.

Delay
(P)

`Delay(anzahl)`
erzeugt eine Verzögerungsschleife von einer »anzahl« Millisekunden.
Beispiel: `delay(12000);`
Ergebnis: Das Programm hält an dieser Stelle für ca. 12 Sekunden an.

Delete
(P)

`Delete(string,anfang,anzahl)`
entfernt aus einem »string« eine »anzahl« Zeichen, beginnend mit einer »anfang« position.
Beispiel:
```
string:='Mülleimer';
Delete(string,5,50);
Write(string);
```
Ergebnis: »Müll«
»anfang« muß innerhalb 0..255 liegen. Es werden nur Zeichen innerhalb des Strings gelöscht.

EOF
(F)

`EOF(file)`
Wird mit True bzw. False belegt, wenn das Ende einer Datei erreicht bzw. nicht erreicht ist.
Beispiel: `IF EOF(orte)= true THEN Write('Ende');`
Ergebnis: Bei Erreichen des letzten Bytes der Datei wird »Ende« ausgegeben. Dieses letzte Zeichen war dann ein CTRL/Z.

EOL
(F)

`EOL(file)`
nimmt True oder False an, wenn das Ende einer Zeile, also ein CTRL/M (Carriage Return) aufgenommen wurde.
Beispiel: entsprechend wie EOF

Erase
(P)

`Erase(file)`
löscht die Datei, die der Variablen »file« zugewiesen wurde, von Diskette.
Beispiel: `Erase(datei1);`

Execute
(P)

`Execute(comfile)`
lädt und startet ein mit **<C>** kompiliertes Programm. Zuvor ist Assign auszugeben.
Beispiel:
```
Assign(comfile,'tabelle.com');
Execute(comfile);
```

Exit
(P)

`Exit`
führt zum Aussprung aus einem Unterprogramm. Bei Aufruf vom Hauptprogramm aus wird dieses beendet.

Exp
(F)

`Exp(zahl)`
berechnet den Potenzwert von e hoch »zahl«.
Beispiel: `x:=Exp(4)`
Ergebnis: x=54.59815003

FilePos
(F)

`FilePos(file)`
gibt die momentane Position des Zeigers aus, der in eine geöffnete Datei zeigt.
Beispiel: `zeigerstand:=FilePos(orte);`

FileSize
(F)

`FileSize(file)`
gibt die Anzahl der in einer Datei »file« enthaltenen Komponenten aus.
Beispiel: `dateilaenge:=FileSize(orte);`

FillChar
(P)

`FillChar(variable,anzahl,zeichen)`
belegt in einer Variablen eine »anzahl« Bytes mit einem bestimmten »zeichen«.
Beispiel:
```
test:='  Text';
FillChar(test,1,'X');
Write(test);
```
Ergebnis: »X Test«

Frac
(F)

`Frac(dezimalzahl)`
gibt den Nachkommastellenteil einer »dezimalzahl« an.
Beispiel: `rest:= Frac(12.0233);`
Ergebnis: rest = 0.0233

GetMem
(P)

`GetMem(zeigervariable, anzahl)`
reserviert einer »zeigervariablen« eine bestimmte »anzahl« Bytes im sogenannten Heap.

GotoXY
(P)

`GotoXY(spalte,zeile)`
setzt den Cursor in eine bestimmte »spalte« und eine bestimmte »zeile«.
Beispiel: `GotoXY(40,10);`

Halt
(P)

`Halt`
beendet ein Programm.

Hi
(F)

`Hi(ganzzahl)`
liefert als Ergebnis das Hi-Byte der »ganzzahl« als neue Integerzahl.
Beispiel:
```
x:= Hi(32768);
Write(x:5);
```
Ergebnis: « 128«, weil 32768=$8000 mit Lo=0 und Hi=$80.

IOresult
(F)

`IOresult`
liefert den Wert Null, wenn bei Ein-/Ausgabe-Operationen kein Fehler aufgetreten ist. Andere Meldungen bedeuten eine bestimmte Fehlerart.
Beispiel:
```
(*$I-*) Reset(file) (*I+*);
IF NOT IOresult=0 THEN
Write('Datei nicht vorhanden');
```
Ergebnis: Warnung bei nicht existierender Datei.

InsLine
(P)

`InsLine`
fügt an der momentanen Cursorposition eine Zeile ein. Die darunterliegenden Zeilen werden nach unten geschoben.

Insert
(P)

`Insert(zusatz,vorhandenen,position)`
fügt in einen »vorhandenen« String ab einer bestimmten »position« einen »zusatz« ein. Maximale Länge 255.
Beispiel:
```
tex:='Inhalt der Fläche';
Insert('neuen ',tex,12);
Write(tex);
```
Ergebnis: »Inhalt der neuen Fläche«

Int `Int(dezimalzahl)`
(F) liefert den Ganzzahlanteil eines dezimalen Zahlenausdrucks als Realzahl.
Beispiel: `zahl:=pi;`
`Write(Int(zahl):3:1);`
Ergebnis: »3.0« (Typ Real)

KeyPressed `KeyPressed`
(F) liefert »true«, wenn eine Taste gedrückt wurde.
Beispiel: `REPEAT UNTIL KeyPressed;`
`IF KeyPressed=true THEN Write('JA');`
Ergebnis: Warteschleife bis Tastendruck, dann »JA«.

Length `Length(string)`
(F) liefert die Anzahl der Zeichen von String als Integerzahl.
Beispiel: `x:=Length('Test')`
Ergebnis: x=4

Ln `Ln(Zahl)`
(F) liefert den natürlichen Logarithmus einer Zahl.
Beispiel: `Y:=Ln(100)`
Ergebnis: Y=4.605170186

Lo `Lo(integerzahl)`
(F) gibt von einer Integerzahl den Lo-Anteil aus.
Beispiel: `I:=Lo(4000)`
Ergebnis: I=160, weil 4000=$0fa0 und $a0=160

Mark `Mark(zeigervariable)`
(P) soll eine Zeigervariable markieren (?)
Beispiel: `Mark(Pointer1)`

Mem `Mem[adresse]`
(Array) spricht ein Element des Speichers an.
Beispiel: `x:=Mem[$FC00]`
Ergebnis: x enthält die Belegung von Adresse $FC00

MemAvail `MemAvail`
(F) liefert den verfügbaren Heap-Bereich in Bytes
Beispiel: `x:=MemAvail`
Ergebnis: x ist Anzahl verfügbarer Bytes im Heap

Move
(P)

`Move(quellvariable,zielvariable,anzahl)`
kopiert eine Anzahl Bytes ab einer Quellvariablen auf die Zielvariable, beginnend bei deren Anfangsadresse.
Beispiel: `Move(speicher1,puffer,500)`
Ergebnis: 500 Byte werden von speicher1 ab auf puffer und die folgenden Bytes übertragen.

New
(P)

`New(zeigervariable)`
legt eine neue Zeigervariable im Heap an.

Odd
(F)

`Odd(integerzahl)`
prüft eine Integerzahl auf ungerade Zahl.
Beispiel: `IF Odd(5) THEN Text:='ungerade'`
Ergebnis: Text='ungerade'

Ord
(F)

`Ord(variable)`
gibt die Ordnungszahl (Zählbeginn bei Null) einer Variablen oder eines Zeichens aus.
Beispiel: `X:=Ord('A')`
Ergebnis: X=65

Port
(Array)

`Port[integerzahl]`
spricht einen Datenport des Prozessors an.
Beispiel: `x:=Port[98]`
Ergebnis: x enthält Belegung des Datenports 98.

Pos
(F)

`Pos(suchstring,zielstring)`
gibt die Position des Suchstrings in einem Zielstring aus. Nicht vorhandener Suchstring erzeugt den Wert 0.
Beispiel: `X:=Pos('wo','Einwohner')`
Ergebnis: X=4

Pred
(F)

`Pred(element)`
gibt den Vorgänger von Element aus einer geordneten Menge aus.
Beispiel: `x:=Pred('Y')`
Ergebnis: x='X'

Ptr
(F)

`Ptr(zeigervariable)`
weist einer Zeigervariablen einen festen Wert zu.
Beispiel: `schirmanfang:=Ptr($4000)`
Ergebnis: schirmanfang zeigt auf $4000

Random (F)
`Random`
erzeugt eine beliebige Zufallszahl, die zwischen 0 (einschließlich) und 1 (ausschließlich) liegt.
`Random(integerzahl)`
erzeugt eine Integer-Zufallszahl, die zwischen 0 und der angegebenen Integerzahl liegt.
Beispiel: `z:=Random(49)`

Randomize (P)
`Randomize`
startet den Zufallsgenerator neu.

Read (P)
`Read(datei,variable)`
liest von einer Eingabedatei Zeichen in die Variable ein. Fehlt der Parameter Datei, wird Input angenommen, im Normalfall die Konsole (über die Tastatur).

ReadLn (P)
`ReadLn(datei,variable)`
liest wie Read, überspringt aber abschließend das Zeilenendezeichen und stellt sich auf den Anfang der nächsten Zeile ein.

Release (P)
`Release(zeigervariable)`
stellt den Heap-Zeiger auf die Zeigervariable ein.

Rename (P)
`Rename(datei,neuer Name)`
benennt eine Datei um.
Beispiel: `Rename(alt,'aktuell');`
Ergebnis: Die unter der Variablen »alt« geführte Datei erhält den neuen Namen »aktuell«.

Reset (P)
`Reset(datei)`
eröffnet eine bestehende Datei zum Schreiben oder Lesen.

Rewrite (P)
`Rewrite(datei)`
eröffnet eine neue Datei.
Ergebnis: alte Datei wird mit nachfolgenden Write-Anweisungen überschrieben.

Round (F)
`Round(realzahl)`
rundet eine Realzahl auf eine Ganzzahl (Typ Integer).
Beispiel: `x:=Round(Pi)`
Ergebnis: x=3 (Typ Integer)

Seek
(P)
`Seek(datei,nummer)`
bewegt den Dateizeiger auf die Komponente mit der angegebenen Nummer (beginnend bei Null).

SeekEoF
(P)
`SeekEoF(datei)`
sucht nach dem Ende der Datei. Alle Tabulatoren, Zeilenendemarkierungen und Leerzeichen werden übersprungen.

SeekEoln
(P)
`SeekEoln(datei)`
sucht nach dem Zeilenende, überspringt auch Tabulatoren und Leerzeichen.

SizeOf
(F)
`SizeOf(bezeichner)`
gibt die Anzahl Bytes aus, die eine Variable »bezeichner« oder ähnlich im Speicher belegt.

Sin
(F)
`Sin(zahl)`
gibt den Sinuswert von »zahl« (Bogenmaß) aus.
Beispiel: `x:=Sin(PI/2)`
Ergebnis: x=1

Sqr
(F)
`Sqr(zahl)`
bildet das Quadrat einer Zahl.
Beispiel: `x:=Sqr(25)`
Ergebnis: x=625

Sqrt
(F)
`Sqrt(zahl)`
bildet die Quadratwurzel aus einer Zahl.
Beispiel: `x:=Sqrt(289)`
Ergebnis: x=17

Str
(P)
`Str(zahl,stringvariable)`
wandelt einen Zahlenwert in einen String um.
Beispiel: `Str(22.5:4,text)`
Ergebnis: text='22.5'

Succ
(F)
`Succ(element)`
gibt den Nachfolger eines Elements in einer geordneten Menge aus.
Beispiel: `X:=Succ('X')`
Ergebnis: X='Y'

Swap `Swap(integerzahl)`
(F) vertauscht Lo- und Hi-Anteil einer Integerzahl.
Beispiel: `X:=Swap(4000)`
Ergebnis: X=40975, weil 4000=$0fa0 und $a00f=40975

Trunc `Trunc(realzahl)`
(F) gibt den Integeranteil einer Realzahl aus.
Beispiel: `X:=Trunc(-Pi)`
Ergebnis: X=-3 (Typ Integer)

UpCase `UpCase(zeichen)`
(F) gibt den entsprechenden Großbuchstaben eines Zeichens aus, falls es existiert.
Beispiel: `X:=UpCase('z')`
Ergebnis: X='Z'

Val `Val(string,zahl,kontrolle)`
(P) wertet die Ziffernfolge eines Strings aus. Die Kontrollzahl weist auf unauswertbare Zeichen hin.
Beispiel: `Val('14.55',Wert,c)`
Ergebnis: Wert=14.55; c=0;

Write `Write(datei,zeichen)`
(P) schreibt Zeichen in eine Datei. Fehlt die Angabe der Datei, wird die Standard-Datei Output, im Normalfall der Bildschirm angenommen.

WriteLn `WriteLn(datei,zeichen)`
(P) schreibt wie oben, fügt aber ein Zeilenende-Zeichen an und beginnt somit eine neue Zeile.

C

Turbo-Compiler-Befehle unter CP/M 80

Die erstgenannte Compileranweisung ist immer der voreingestellte Befehl. Der Hinweis »schneller!« gibt an, daß das Programm mit dieser Einstellung beschleunigt abläuft. Solche Einstellungen sollten nur benützt werden, wenn das Programm fehlerfrei läuft.

{$A+ }	Code-Erzeugung ohne Rekursion (schneller!)
{$A- }	Code-Erzeugung, die Rekursion erlaubt.
{$B+ }	CON-Gerät ist INPUT/OUTPUT-Datei
{$B- }	TRM-Gerät ist INPUT/OUTPUT-Datei
{$C+ }	<CTRL/C> unterbricht Programmlauf bei Read, ReadLn
{$C- }	keine Interpretation von Kontrollzeichen (schneller!)
{$I Datei }	Include-Anweisung für eine externe Datei.
{$I+ }	I/O-Fehlerüberprüfung aktiv
{$I- }	I/O-Fehlerüberprüfung ausgeschaltet (schneller!)

{$R+ }	Indexprüfung beim Programmlauf eingeschaltet.
{$R- }	keine Indexprüfung (schneller!)
{$U+ }	Programmunterbrechung mit **<CTRL/C>** immer möglich. (Bereitet beim CPC 6128 gelegentlich Schwierigkeiten!)
{$U- }	keine Programmunterbrechung zugelassen (schneller!)
{$V+ }	Typprüfung von Strings bei Variablenparametern.
{$V- }	keine Typprüfung (schneller!).
{$WX }	Einstellung der Schachtelungstiefe für With-Anweisungen. Voreinstellung: X=2 ; maximal X=9.
{$X+}	Optimiert die Code-Erzeugung bezüglich der Array-Verwaltung auf maximale Geschwindigkeit.
{$X-}	Optimiert die Array-Verwaltung auf minimalen Code.

D

Tastatur-Codes

Tastendrücke lassen sich mit Turbo folgendermaßen identifizieren:

- Einfacher Tastendruck erzeugt einen einfachen Code.
- Kombinierte Tasten erzeugen in den meisten Fällen andere Codewerte.

Das Haupttastenfeld:

Taste	<einfach>	<mit SHIFT>	<mit CONTROL>
ESC	252	252	252
TAB	9	9	225
RETURN	13	13	13
A	97	65	1
B	98	66	2
C	99	67	3
D	100	68	4
E	101	69	5

Taste	<einfach>	<mit SHIFT>	<mit CONTROL>
F	102	70	6
G	103	71	7
H	104	72	8
I	105	73	9
J	106	74	10
K	107	75	11
L	108	76	12
M	109	77	13
N	110	78	14
O	111	79	15
P	112	80	16
Q	113	81	17
R	114	82	18
S	115	83	19
T	116	84	20
U	117	85	21
V	118	86	22
W	119	87	23
X	120	88	24
Y	121	89	25
Z	122	90	26
.	46	62	--
,	44	60	--
/	47	63	--
Ö	92	96	28
:	58	42	--
;	59	43	--
Ü	93	125	29
§	64	124	0
Ä	91	123	27
-	45	61	--
^	94	126	30
CLR	16	16	16
DEL	127	127	127

Taste	<einfach>	<mit SHIFT>	<mit CONTROL>
0	48	95	31
1	49	33	--
2	50	34	126
3	51	35	--
4	52	36	--
5	53	37	--
6	54	38	--
7	55	39	--
8	56	40	--
9	57	41	--

Numerisches Tastenfeld:

Taste	<einfach>	<mit SHIFT>	<mit CONTROL>
1	49	49	49
2	50	50	50
3	51	51	51
4	52	52	52
5	53	53	53
6	54	54	54
7	55	55	55
8	56	56	56
9	57	57	57
0	48	48	48
Pfeil auf	240	244	248
Pfeil ab	241	245	249
Pfeil li	242	246	250
Pfeil re	243	247	251
.	46	46	46
ENTER	13	13	82

<CTRL/S> stoppt Ausgabe
<CTRL/SHIFT/ESC> bewirkt einen Reset

Anhang

E

Bedienung des Turbo-Editors

Cursorsteuerung

<CTRL/A>	zum Anfang des linksstehenden Worts
<CTRL/H>	ein Zeichen nach links
<CTRL/S>	ein Zeichen nach links
<CTRL/D>	ein Zeichen nach rechts
<CTRL/F>	zum Anfang des rechtsstehenden Worts
<CTRL/E>	Cursor um eine Zeile nach oben
<CTRL/X>	Cursor um eine Zeile nach unten
<CTRL/W>	Bildschirm um eine Zeile nach unten rollen
<CTRL/Z>	Bildschirm um eine Zeile nach oben rollen
<CTRL/R>	Bildschirm um eine Seite zurückblättern
<CTRL/C>	Bildschirm um eine Seite vorblättern
<CTRL/Q><S>	Cursor an den Zeilenanfang
<CTRL/Q><D>	Cursor an das Zeilenende

<CTRL/Q><E>	Cursor zum Bildschirmanfang (HOME)
<CTRL/Q><X>	Cursor zur letzten Bildschirmzeile
<CTRL/Q><R>	Cursor an den Textanfang
<CTRL/Q><C>	Cursor an das Textende
<CTRL/Q><P>	Cursor zur vorhergehenden Position (Memory)
<CTRL/Q><B>	Cursor zum Blockanfang
<CTRL/Q><K>	Cursor zum Blockende

Löschen und Einfügen

<CTRL/V>	Einfügemodus Ein/Aus (Wechselschalter)
<CTRL/N>	Leerzeile einfügen
<DEL>	Zeichen links vom Cursor löschen
<CTRL/G>	Zeichen an der Cursorposition löschen
<CTRL/T>	Wort rechts vom Cursor löschen
<CTRL/Y>	Cursorzeile löschen
<CTRL/Q><Y>	rechtes Zeilenende ab Cursorposition löschen

Blockbearbeitung

<CTRL/K><B>	Markieren des Blockanfangs
<CTRL/K><K>	Markieren des Blockendes
<CTRL/K><T>	Markieren eines Wortes als Mini-Block
<CTRL/K><H>	Block revers/normal (Wechselschalter)
<CTRL/K><C>	Kopieren des markierten Blocks an die Cursorposition
<CTRL/K><V>	Verschieben eines Blocks an die Cursorposition
<CTRL/K><Y>	Löschen des markierten Blocks
<CTRL/K><W>	Speichern des markierten Blocks auf Diskette
<CTRL/K><R>	Laden eines Blocks von Diskette

Diverse Kommandos

<CTRL/I>	Tabulator
<CTRL/Q><F>	Suchfunktion einschalten
<CTRL/Q><A>	Suchen und Ersetzen einschalten
<CTRL/L>	letzte Suche wiederholen
<CTRL/P>	Kontrollzeichenwahl
<CTRL/K><D>	Editor verlassen, zurück ins Turbo-Menü

Auch auf den Pfeiltasten sind Editierfunktionen implementiert:

links	ein Zeichen nach links
SHIFT/links	an den Zeilenanfang
CTRL/links	ein Wort nach links
rechts	ein Zeichen nach rechts
SHIFT/rechts	an das Zeilenende
CTRL/rechts	ein Wort nach rechts
oben	eine Zeile nach oben
SHIFT/oben	Schirm nach oben scrollen
CTRL/oben	eine Seite zurückblättern
unten	eine Zeile nach unten
SHIFT/unten	Schirm nach unten scrollen
CTRL/unten	eine Seite weiterblättern
COPY	wie <CTRL/V>

Anhang

F

Literatur

1. Turbo-Pascal 3.0 Compiler Handbuch, Borland International

2. CP/M PLUS Betriebssystem, Markt und Technik

3. Programmer's Utilities and SID für CP/M.., Markt und Technik

4. Turbo Pascal Program Library, QUE Corp., Indianapolis

5. ROM-Listing CPC, Markt und Technik

6. CP/M PLUS Anwenderhandbuch, Markt und Technik

7. Programmierung des Z80, Sybex

8. Programmieren mit Pascal, CHIP

9. Einführung in die Programmierung mit Pascal und UCSD-Pascal, Sybex

Stichwortverzeichnis

A

Abbruchbedingung 45, 102
Absolute 302
absolute Variablen 339
Abspeichern 238
abzählbare Ordnung 266
Addr 160, 204, 206
Adresse 161
Algorithmus 97, 124
AND 52, 89, 282, 284
Anfangsadresse 309
anonyme Variablen 157
Anweisungen 45
Anweisungsteil 36
Arbeitsdiskette 26
ArcTan 80, 88
arithmetische Funktionen 105
Array 43, 107, 116, 157, 219, 262, 266, 309
Array-Konstante 114, 128
ASCII 137, 139, 170, 196, 203, 232
Assembler 20, 279, 302, 303
Assign 223, 256, 260, 263, 336
Ausgabe 42
Aux 229, 256
AuxOutPtr 229

B

Background 177
Backspace 57
Bank 280, 316, 321
Bankswitching 321
BASIC 19, 20, 117, 118
BDOS 279, 291, 294
BDOSHL 293
Bedingung 52, 53
BEGIN 42
Begrenzer 111
Betriebssystem 277, 278, 317, 320
Betriebssystemroutine 288
Bildschirm 26
Bildschirmanfang 317
Bildschirmauflösung 317
Bildschirmbyte 320
Bildschirmdaten 328
Bildschirminhalt 316

Bildschirmmaske 190
Bildschirmmodus 169, 174, 179
Bildschirm-Programme 325
Bildschirm-RAM 318
Bildschirmseite 316, 324
Bildschirmspeicher 281, 316, 322
Bildschirmverwaltung 316
BIOS 279, 304
BIOS-Aufrufe 298, 304
BIOS-Nr. 295
BIOS-Routinen 295
Bits ausschalten 284
Bitverknüpfungen 283
Block 256, 260, 280
Blockbefehle 84
BlockRead 263, 302
BlockWrite 260, 261, 263
Bogenmaß 82
Boolean 228, 309
Bubble-Sort 231, 232
BufLen 41, 190, 222
Byte 309
Byte-Array 325

C

CASE ... OF 143, 199
Chain 256, 278, 334, 337, 338, 339
Char 51, 137, 140, 309
Characters 51
Chr 199, 200
Close 225, 263
ClrEOL 225
ClrScr 33
Code 51
Code-Nummer 199
Codierung 109
Compiler 66, 75, 254, 329, 331, 333
Compilerbefehl 223
Compiler-Optionen 331, 338
CON 256
Concat 38
ConOutPtr 229
CONST 42, 118
Copy 57, 58, 151
Cos 79
CP/M 279, 281, 287, 292, 337
Cursor 40, 175, 191, 196, 291

D

Datei 211
Dateiende 252
Dateinamen 223
Datei, typlose 256, 263
Dateivariablen 223
Dateizeiger 212, 226, 227, 238, 239, 255, 257
Daten, skalare 122
Daten, strukturierte 107
Datenkanäle 212
Datensatz 236
Datentyp 41, 123
Datentypen 63, 213
Datentypen, frei definierte 117
Deklarierung 36
Deklarierungsteil 37
Delete 151, 200
DelLine 45, 218
Dezimaldarstellung 59
Dezimalsystem 164
Dimensionierung 109, 111
Directory-Eintrag 296
Directory-Tafel 294
Disassembler 281
DISK RESET 294
Disk-Reset 293
Diskettendatei 248
Diskettensektor 296
Dispose 159
DMA-Puffer 292
DO 112
DOWNTO 111
Drive-Anzeige 175, 258
Drucker 229, 231, 256
dynamische Variablen 156

E

Echo 39, 189
Eingabe-Echo 189
Eingabekommentar 195
Eingaben 40, 202
Eingabepuffer 40, 41
Einsprünge 289
Einsprungadresse 287, 304
Element 271
Elemente 137, 265

ELSE 74, 220
END 46
ENTER-Taste 24
EOF 228, 252
EOLN 254
Erase 240, 256
Erweitern 249
ESC-Funktionen 170, 181
Execute 256, 278, 316, 334, 336, 338, 339
Exit 143
Exklusiv Oder 283
Exp 102, 103, 104
Exponentialdarstellung 59
External 277, 299, 309, 310, 312, 314, 318
External-Funktion 313
External-Prozeduren 308
External-Routinen 324

F

False 134, 232
Farbe 177
Farbskala 178
Farbwechsel 181
Farbwerte 177
Feld 43, 45
Feldelemente 45, 113
Feldkonstante 127
Feldvariable 44, 129
Fenster 189
Festwertparameter 56, 309, 312
File 212, 256, 263, 307, 336
File Of 212, 237, 242
File-Dateien 256
FilePos 213, 242, 248, 257
FileSize 213, 227, 238, 242, 248, 257
FillChar 202, 204, 206, 207, 210
Flush 213
FOR 111, 116, 145
FOR-Anweisungen, geschachtelte 117
FOR-Schleife, geschachtelte 148
formatieren 42
formatiert abspeichern 250
Formatierung 59
FORWARD 63
Frac 72
FreeMem 160
FUNCTION 78
Funktionen 77, 80, 85, 87, 95
Funktionen, arithmetische 105
Funktionen, ESC- 170
Funktionen, lokale 81
Funktionen, rekursive 97
Funktionsergebnis 314
Funktionsnummer 292

G

Ganzzahlen 45
Gate Array 281
geschachtelt 117, 148
geschachtelte Schleife 126
GetMem 158, 159, 261
Gleichung, quadratische 274
global 48
globale Variablen 150, 339
Goto 143
GotoXY 40, 130, 169, 193
Grafik 286, 289
Grafikschirm 291
Grafikseite 324
Grundmenge 268

H

Halt 102, 103
Hauptmenü 75
Heap 95, 156, 157
Heap-Zeiger 158, 160
Heap-/Stack-Collision 95
HeapPtr 96, 158
Hex-Dump 282
Hex-String 284
Hexzahlen 283, 305
Hi 284
Hintergrundfarbe 177

I

IF 46
IF ... THEN 45
In 138
Include-Anweisung 66, 81, 277
Include-Dateien 66, 75, 168, 332

Index 108, 113, 118, 133
indizierte Variablen 108
Inhaltsverzeichnis 302, 330, 332, 335
Initialisierung 37, 142, 150, 193
Inline 312, 320
Inline-Anweisung 277, 285
Inline-Funktion 314
Inline-Programme 310
Input 39, 212
Insert 201
Installieren 26
Int 74
Integer 45, 219, 309
IntegerArray 193
Integermengen 272
Integerzahl 161
Intervall 268
IOresult 223
irrationale Zahlen 97
Iteration 100

K

KBD 52, 140, 194, 256
KeyPressed 134, 336
Klammern 267, 268, 276
Kommentarklammern 37, 130
Kommentierung 23
Kompilieren 20, 35, 333, 336, 339
Komplementärmenge 269, 271
Komponenten 108, 113, 128
Konsole 256
Konstante 37, 325
Konstante, Array- 114, 128
Konstante, typisierte 37, 38, 114
Kontrollausdrucke 69
Kontrollvariablen 59
Kontrollzeichen 70
Koordinaten 67, 291
Korrigieren 236
Kosinus 79
Kursberechnung 87

L

Label 142
Labelbezeichner 142
Laden 251
Länge des Strings 204
Laufwerk 293
Leerzeichen 38
Length 57
Library 20, 334
Ln 102, 103, 104
Lo 284, 311
Löschen 239
Logarithmus, natürlicher 103
LST 256, 285
LstOut 229
LstOutPtr 229

M

Manipulation mit Bits 284
Maschinenbefehle 290, 302
Maschinencode 285, 305, 312
Maschinenprogramm 277, 285, 302, 308, 324,
Maschinenprogramme, externe 299
Maschinensprache 20, 302
Maske 189, 191
Matrix 317
Matrizen 112
Mem 207, 209
MemAvail 160
Menge 56, 136, 265, 309
Mengendeklarierungen 267
Mengenkonstanten 267
Mengenlehre 265
Mengenverknuepfungen 268
Menü 135, 143, 216
Menütafel 143
Modus 181
Moduswechsel 175
Monitor 281
Move 207

N

natürlicher Logarithmus 103
New 157, 159, 261
Nil 163
Normen 191
NOT 52, 282

O

Obermenge 271, 276
Objektcode 305, 306
Option 75, 333, 338, 339
OR 53, 282
Ord 124, 199, 267
Ordnung, abzählbare 266
Ordnungszahl 44, 109, 127, 272
OUT 281
Output 39, 212, 227
Overlay 278, 329
Overlay-Block 329, 331, 332
Overlay-Datei 331, 332, 333
OvrDrive 334

P

Parameter 49, 62, 310
Parameterübergabe 308
Permutation 133
PI 82
Pixel 317
Platzersparnis 333
PLOT 287
Pointer 156
POP 310
PROCEDURE 48
PROFILE.SUB 28
Prozedur 39, 47, 87, 217
Prozedur, rekursive 97
Prozeduren, verschachtelte 177
Prozedurparameter, formaler 54
Prozessor 278
Ptr 161, 206, 207
Puffer 258, 295, 298, 299, 300, 320, 321, 328
PUSH 310

Q

quadratische Gleichung 274
Quadratwurzel 97
Quelltext 303

R

RAD 82
Radikand 97
RAM-Vektoren 288
Random 147, 151
Randomize 147
Rasterzeilen 317
Read 38, 41, 190, 242
ReadLn 41
Real 157
Realzahl 161
Realzahlen 55
Rechenarten 104, 105
Record 107, 218, 219, 221, 235, 266, 309
RecurPtr 96
Reelle Zahlen 309
Register 279
Registerbelegung 295
Rekursion 80, 95, 97, 102, 158
Rekursionsstack 96
Release 159
Rename 256
REPEAT 45, 52, 140, 225, 254
REPEAT ... UNTIL 134, 225
Reset 96, 223, 225, 239, 249, 256, 263, 293
RETURN-Taste 24
Retyping 127
revers 241
reverse Schrift 215
Rewrite 226, 239, 249, 256, 260, 263
Ringtausch 133
Round 74
Rücksprungadresse 309, 310
Runtime 337

S

Scheinprozedur 333
Schirm löschen 258, 262
Schirmanfang 322
Schleife 45, 56, 116, 129, 225, 234, 235
Schleife, geschachtelte 126
Schnittmenge 269, 270
Schnittstelle 256
Schreibweise 69, 111
Schrittweite 111
SCR BASE 316
SCR OFFSET 317
Seek 213, 226, 238, 242, 248, 257
Sektor 298, 299, 303
Semantik 23

Set Of 51, 137, 266
Shiften 284
SHL 282
SHR 282
skalare Daten 122
Sonderfunktion 170
Sortierverfahren 231
Sound 289
Spalte 40
Speicheradressen 202
Speicherbereich 280
Speicherplatz, freier 294, 335
Speicherverwaltung 280
Sprungtabelle 304
Spur 296, 298
Sqr 64
Sqrt 79
Stack 95, 156, 309, 310
Stackpointer 309, 312
StackPtr 96
Stackverwaltung 310
Standard-Dateien 39, 227
Standard-Zeichensatz 177, 214
Stapel 156
Startadresse 338
statische Variablen 156, 157
Steuerdaten 231
Str 129, 130
Streckenberechnung 81
String 37, 157, 309
String-Array 194
Stringinhalte 202
Stringlänge 204
Stringverknüpfung 38
strukturierte Daten 107
SUBMIT.COM 28
Syntax 23
Systemadresse 281

T

Tastatur 52, 140, 194, 256
Tastaturabfrage 142
Tasten 24
Teillösungsmenge 269
Teilmenge 271, 276
Terminal 256
Text 117, 212, 242, 247, 256
Text-Dateien 255
Textschirm 291
Trennzeichen 247, 286
TRM 256
True 134, 228, 232
Turbo-Editor 303
Turbo-Hauptmenü 75
Turbo-Library 337
Turbo-Menü 135, 334
Turbo-Wörter 37
Typbezeichner 54
TYPE 42
Typerklärung 42
typisierte Konstante 114
typlose Datei 256, 263
Typüberprüfung 254

U

Uhr 155
Umlaute 196
Ungleichheit 271
Ungleichungen 268
Unterprogramm 329, 333
UNTIL 45, 225
UpCase 139, 235
Usr 256
UsrOutPtr 229

V

Val 57, 59, 74
VAR 42
Variablen 37, 39, 48, 267
Variablen, absolute 339
Variablen, anonyme 157
Variablen, dynamische 156
Variablen, globale 48, 51, 150, 339
Variablen, indizierte 108
Variablen, lokale 48
Variablen, statische 156, 157
Variablen, Zeiger- 157
Variablenadresse 207
Variablenparameter 56, 60
Vereinigungsmenge 270
Verkettungen 278
Verknüpfungen 52, 54

verschachtelte Prozeduren 177
Video-RAM 316
Vorzeichen 59

W

wahlfreier Zugriff 212
Warteschleife 164, 336
Wechselwertparameter 310
Wert 39
Wertparameter 60
Wertzuweisung 40
WHILE ... DO 140, 220
With 221, 222
Write 33, 37, 242
WriteLn 37
Würfelspiel 168

X

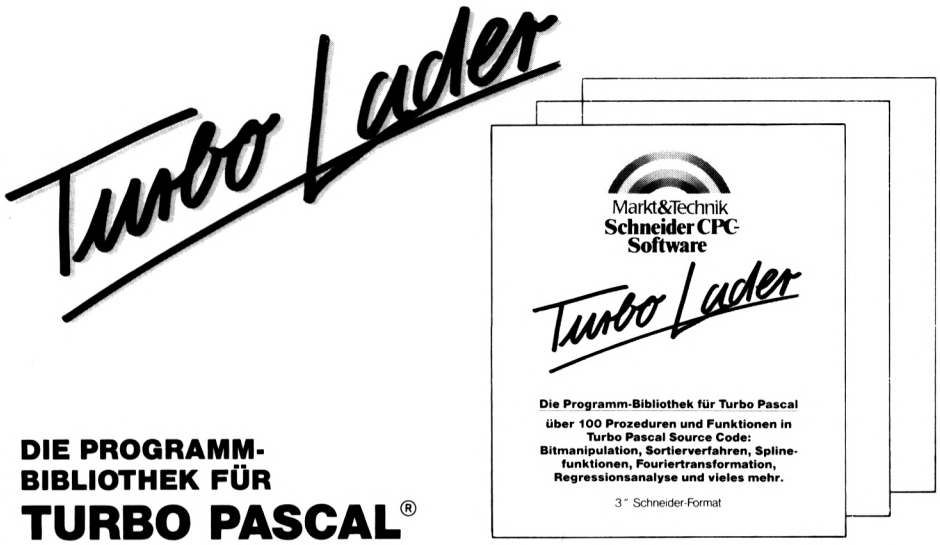
XOR 283

Z

Zähler 112, 118, 124, 129, 165
Zählwerk 164
Zahlen, irrationale 97
Zauberwürfel 121
Zeichen 136, 196
Zeichenfarbe 177
Zeichenkette 37, 148
Zeichensatz 176, 177, 179, 317
Zeichensatz, deutscher 179, 214
Zeichensatz, Standard- 177
Zeiger 156, 207, 309
Zeigeroperationen 167
Zeigervariablen 156, 157, 261, 298
Zeile 40
Zeitmessung 163
Zufallsgenerator 150
Zufallszahlen 147
Zugriff, wahlfreier 212
zuordnen 235
Zuordnung 77, 88
zuweisen 221
Zuweisung 37, 39, 126, 129, 138, 161, 166,
Zuweisungsschleife 129

{$A-} 97, 101
{$I+} 224
{$I-} 224
{$V+} 254
{$V-} 254
{$WX} 223

JETZT AUF SCHNEIDER-COMPUTERN:



DIE PROGRAMM-BIBLIOTHEK FÜR TURBO PASCAL®

TURBO-Lader-Grundpaket

Das TURBO-Lader-Grundmodul ist eine umfangreiche Programm-Bibliothek für den TURBO-Pascal-Programmierer. Sie umfaßt zahlreiche ausführlich dokumentierte Prozeduren und Funktionen, die der Profi zur schnellen Lösung seiner Programmieraufgaben verwenden kann.

Das TURBO-Lader-Grundpaket erfordert den TURBO-Pascal-Compiler. Es ist lieferbar auf 3"- und 5¼"-Disketten und lauffähig auf dem Schneider CPC 464, CPC 664, CPC 6128.

3"-Disk. Best.-Nr. MS 413
5¼"-Disk. Best.-Nr. MS 415

DM 138,-* * inklusive MwSt.

TURBO-Lader Business

TURBO-Lader Business umfaßt einen komfortablen Bildschirm-Maskengenerator und eine professionelle Dateiverwaltung. Der Maskengenerator gibt dem Pascal-Programmierer ein Werkzeug zur einfachen Bearbeitung von Bildschirm-Masken in die Hand.

TURBO-Lader Business erfordert den TURBO-Pascal-Compiler und das TURBO-Lader-Grundpaket. Es ist lieferbar auf 3"- und 5¼"-Disketten und lauffähig auf dem Schneider CPC 464, CPC 664, CPC 6128.

3"-Disk. Best.-Nr. MS 423
5¼"-Disk. Best.-Nr. MS 425

DM 148,-* * inklusive MwSt.

TURBO-Lader Science

TURBO-Lader Science ist eine Sammlung technisch/wissenschaftlicher Funktionen und professioneller statistischer Verfahren für die Bereiche Medizin, Betriebs- und Volkswirtschaft, Technik und Naturwissenschaften.

TURBO-Lader Science erfordert den TURBO-Pascal-Compiler und das TURBO-Lader-Grundpaket. Es ist lieferbar auf 3"- und 5¼"-Disketten und lauffähig auf dem Schneider CPC 464, CPC 664, CPC 6128.

3"-Disk. Best.-Nr. MS 433
5¼"-Disk. Best.-Nr. MS 435

DM 189,-* inklusive MwSt.

Übrigens können Sie auch folgende Turbo-Pascal-Produkte für Schneider CPC und Joyce bei Markt&Technik beziehen:
Turbo Pascal 3.0, Turbo Pascal 3.0 mit Grafikunterstützung,
Turbo Tutor (deutsch), Turbo Tutor (englisch), Turbo Graphix Toolbox, Turbo Toolbox.

TURBO Pascal® ist ein Warenzeichen der Borland Inc., USA. TURBO-Lader, TURBO-Lader Business und TURBO-Lader Science sind Warenzeichen der Fa. Lauer & Wallnitz.

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701334

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Spitzen-Software für Schneider-Computer

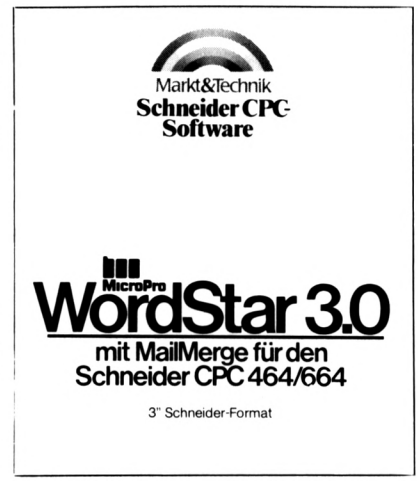
WordStar 3.0 mit MailMerge

Der Bestseller unter den Textverarbeitungsprogrammen für PCs bietet Ihnen bildschirmorientierte Formatierung, deutschen Zeichensatz und DIN-Tastatur sowie integrierte Hilfstexte. Mit MailMerge können Sie Serienbriefe mit persönlicher Anrede an eine beliebige Anzahl von Adressen schreiben und auch die Adreßaufkleber drucken.

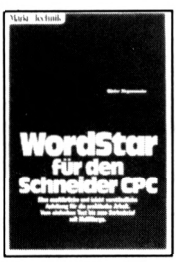
WordStar/MailMerge für den Schneider CPC 464*, CPC 664*
Best.-Nr. MS 101 (3"-Diskette)
Best.-Nr. MS 102 (5¼"-Diskette im VORTEX-Format)
WordStar/MailMerge für den Schneider CPC 6128
Best.-Nr. MS 104 (3"-Diskette)
WordStar/MailMerge für den Schneider Joyce PCW 8256
Best.-Nr. MS 105 (3"-Diskette)

Hardware-Anforderungen: Schneider CPC 464*, CPC 664*, CPC 6128 oder Joyce, beliebiger Drucker mit Centronics-Schnittstelle
* Der Standard-Speicherplatz beim CPC 464/664 erlaubt ohne Speichererweiterung Blockverschiebe-Operationen nur bedingt und Simultan-Drucken gar nicht.

Dieses Programm kostet
DM 199,– inkl. MwSt. Unverbindliche Preisempfehlung



Und dazu die weiterführende Literatur:



Mit diesem Buch haben Sie eine wertvolle Ergänzung zum WordStar-Handbuch: Anhand vieler Beispiele steigen Sie mühelos in die Praxis der Textverarbeitung mit **Wordstar** ein. Angefangen beim einfachen Brief bis hin zur umfangreichen Manuskripterstellung zeigt Ihnen dieses Buch auch, wie Sie mit Hilfe von MailMerge Serienbriefe an eine beliebige Anzahl von Adressen mit persönlicher Anrede senden können.

Best.-Nr. MT 779
ISBN 3-89090-180-8
DM 49,–

Erhältlich bei Ihrem Buchhändler.

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701331

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Spitzen-Software für Schneider-Computer

dBASE II, Version 2.41

dBASE II, das meistverkaufte Programm unter den Datenbanksystemen, eröffnet Ihnen optimale Möglichkeiten der Daten- und Dateihandhabung. Einfach und schnell können Datenstrukturen definiert, benutzt und geändert werden. Der Datenzugriff erfolgt sequentiell oder nach frei wählbaren Kriterien, die integrierte Kommandosprache ermöglicht den Aufbau kompletter Anwendungen wie Finanzbuchhaltung, Lagerverwaltung, Betriebsabrechnung usw.

dBASE II für den Schneider CPC 464*, CPC 664*
Best.-Nr. MS 301 (3"-Diskette)
Best.-Nr. MS 302 (5¼"-Diskette im VORTEX-Format)
dBASE II für den Schneider CPC 6128
Best.-Nr. MS 304 (3"-Diskette)
dBASE II für den Schneider Joyce PCW 8256
Best.-Nr. MS 305 (3"-Diskette)

Hardware-Anforderungen: Schneider CPC 464*, CPC 664*, CPC 6128 oder Joyce, beliebiger Drucker mit Centronics-Schnittstelle
* dBASE II für den Schneider CPC 464/664 ist lauffähig mit einer Speichererweiterung auf 128 Kbyte.

Dieses Programm kostet
DM 199,– inkl. MwSt. Unverbindliche Preisempfehlung

für den Schneider CPC 6128

3" Schneider-Format

Und dazu die weiterführende Literatur:

Zu einem Weltbestseller unter den Datenbanksystemen gehört auch ein klassisches Einführungs- und Nachschlagewerk! Dieses Buch des deutschen Erfolgsautors Dr. Peter Albrecht begleitet Sie mit nützlichen Hinweisen, die nur von einem Profi stammen können, bei Ihrer täglichen Arbeit mit **dBASE II.** Schon nach Beherrschung weniger Befehle ist der Einsteiger in der Lage, Dateien zu erstellen, mit Informationen zu laden und auszuwerten.

Best.-Nr. 90188
ISBN 3-89090-188-3
DM 49,–

Erhältlich bei Ihrem Buchhändler.

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701332

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 46 13-0

Spitzen-Software für Schneider-Computer

Multiplan, Version 1.06

Wenn Sie die zeitraubende manuelle Verwaltung tabellarischer Aufstellungen mit Bleistift, Radiergummi und Rechenmaschine satt haben, dann ist MULTIPLAN, das System zur Bearbeitung »elektronischer Datenblätter«, genau das richtige für Sie! Das benutzerfreundliche und leistungsfähige Tabellenkalkulationsprogramm kann bei allen Analyse- und Planungsberechnungen eingesetzt werden wie z. B. Budgetplanungen, Produktkalkulationen, Personalkosten usw. Spezielle Formatierungs-, Aufbereitungs- und Druckanweisungen ermöglichen außerdem optimal aufbereitete Präsentationsunterlagen!

MULTIPLAN für den Schneider CPC 464*, CPC 664*
Best.-Nr. MS 201 (3"-Diskette)
Best.-Nr. MS 202 (5¼"-Diskette im VORTEX-Format)
MULTIPLAN für den Schneider CPC 6128
Best.-Nr. MS 204 (3"-Diskette)
MULTIPLAN für den Schneider Joyce PCW 8256
Best.-Nr. MS 205 (3"-Diskette)

Hardware-Anforderungen: Schneider CPC 464*, CPC 664*, CPC 6128 oder Joyce, beliebiger Drucker mit Centronics-Schnittstelle
* MULTIPLAN für den Schneider CPC 464/664 ist lauffähig mit einer Speichererweiterung auf 128 Kbyte

Dieses Programm kostet
DM 199,– inkl. MwSt. Unverbindliche Preisempfehlung

Markt&Technik
Schneider CPC-Software

MICROSOFT MULTIPLAN für den Schneider CPC 464

3" Schneider-Format

Und dazu die weiterführende Literatur:

Dank seiner Menütechnik ist **MULTIPLAN** sehr schnell erlernbar. Mit diesem Buch von Dr. Peter Albrecht werden Sie Ihre Tabellenkalkulation ohne Probleme in den Griff bekommen. Als Nachschlagewerk leistet es auch dem Profi nützliche Dienste.

Best.-Nr. MT 835
ISBN 3-89090-186-7
DM 49,–

Erhältlich bei Ihrem Buchhändler.

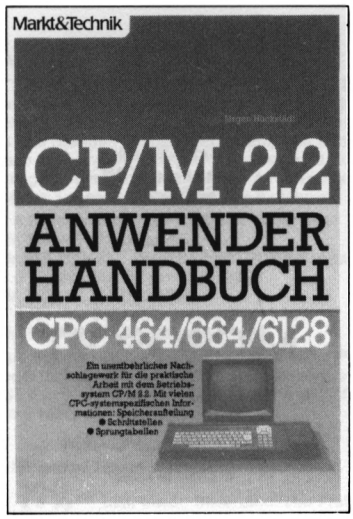
Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701333

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bücher für Schneider-Computer



J. Hückstädt
CP/M-2.2-Anwenderhandbuch CPC 464/664/6128
1985, 212 Seiten

Wenn Sie glücklicher Besitzer eines Schneider-Computers sind und mehr wissen wollen über das leistungsstarke Betriebssystem CP/M 2.2, dann ist dieses Buch genau das richtige für Sie! Es behandelt CP/M 2.2 nicht nur in seiner allgemeinen Form, wie Sie für sämtliche CP/M-Computer gültig ist, sondern bezieht auch die Hardware der CPC-Computer mit ein.

Best.-Nr. MT 859
ISBN 3-89090-204-9
DM 46,-/sFr. 42,30/öS 358,80

C. Straush
Schneider CPC Grafik-Programmierung
1986, 225 Seiten

Dieses Buch wendet sich an die Schneider-CPC-Besitzer, die alles über die Grafikfähigkeiten ihres Computers wissen wollen. Es bietet einen umfassenden Überblick über die verschiedenen Anwendungsbereiche der Grafikprogrammierung: zwei- und dreidimensionale Diagrammdarstellungen, Definition und Bewegung von Sprites, Entwurf von Titelgrafiken, Einsatz der Grafik bei der Unterstützung anderer Programme.

- Besonders interessant: ein Sprite-Generator, ein Malprogramm für hochauflösende Grafik, ein Programm zur Erstellung von Titelgrafiken sowie ein universelles Darstellungsprogramm.

Best.-Nr. 90182
ISBN 3-89090-182-4
DM 46,-/sFr. 42,30/öS 358,80

J. Hückstädt
Der Schneider CPC 6128
1985, 273 Seiten

Dieses Buch ist für jeden CPC-6128-Besitzer eine wertvolle Hilfe, die vielfachen Möglichkeiten dieses bisher einmaligen Computers kennenzulernen und anzuwenden. Der Computerneuling wird Schritt für Schritt in den Umgang mit dem Computer und die BASIC-Programmierung eingeführt, bis er alle notwendigen Kenntnisse besitzt, die mancher Profi bereits mitbringt. Aber an dieser Stelle wird das Programmieren mit dem CPC 6128 erst interessant, nämlich dann, wenn es darum geht, eine eigene Dateiverwaltung aufzubauen oder Grafik und Sound zu programmieren. Weiterhin erfahren Sie alles über CP/M Plus auf dem CPC 6128.

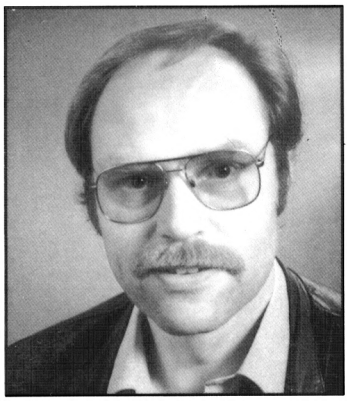
Best.-Nr. 90192
ISBN 3-89090-192-1
DM 46,-/sFr. 42,30/öS 358,80

Markt&Technik-Produkte erhalten Sie in den Fachabteilungen der Warenhäuser, im Versandhandel, in Computerfachgeschäften oder bei Ihrem Buchhändler.

701335

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0



WINFRIED KASSERA,
Jahrgang 1942, Lehrer an der Realschule Neu-Ulm. Ab 1982 Einführung von Commodore-Geräten für das Fach Informatik. Dadurch intensive Beschäftigung mit diesen Rechnern. In der Freizeit leidenschaftlicher Flieger mit Lizenzen für Motor- und Segelflugzeuge. Seit 1966 Fluglehrer und später auch Prüfer. Für den Einsatz in der Ausbildung der Luftfahrer Entwicklung von Simulationsprogrammen als Maschinenprogramme für die 65er-Prozessoren. Lehrbuch für Segelflieger »Flug ohne Motor« bereits in der 8. Auflage. Tätigkeiten als Übersetzer und Lektor für Luftfahrtbücher.

TURBO-PASCAL 3.0 auf dem CPC6128

Turbo-Pascal hat sich der Mängel entledigt, die andere Pascal-Compiler an einer schnellen Verbreitung gehindert haben. Komfortable Programmeditierung, Höchstgeschwindigkeit bei der Erstellung des Objektcodes und beim Programmlauf sind neben größter Zuverlässigkeit bestechende Merkmale.
In diesem Buch wird gezeigt, wie man auf dem Schneider CPC 6128 Turbo-Programme erstellt. Dabei wird der Befehlssatz ausführlich in sinnvollen Beispielen erläutert. Notwendige Grundausstattung ist lediglich ein Schneider CPC 6128 mit monochromem Bildschirm, um alle aufgeführten Programme und Gedanken nachvollziehen zu können. Und natürlich eine Turbo-Pascal-Diskette, möglichst in Version 3.0.

Ausgehend von einfachen Einführungsbeispielen erfahren Sie u. a.:
- wie man mit Turbo-Pascal Rechenprobleme löst
- wie man Bildschirm-Masken erstellt
- wie man ein Menü aufbaut
- wie man ein Spiel entwickelt
- wie man Dateien verwaltet
- wie man Routinen aus dem Betriebssystem anzapft
- wie man das Bankswitching in den Griff bekommt
- wie man Maschinenprogramme integriert
- wie man besondere Eigenschaften von TURBO ausnützen kann

... und vieles mehr.

Auch die Pascal-eigenen Sprachbegriffe kommen nicht zu kurz: Funktionen und Prozeduren werden eingehend behandelt und in Anwendungsbeispielen eingesetzt, so daß jeder, der sich schon einmal mit einer Programmiersprache beschäftigt hat, auch mit Turbo-Pascal bestens zurechtkommen wird. BASIC gerät dabei fast in Vergessenheit.

Hardware-Anforderungen:
- Schneider CPC 6128, monochromer Bildschirm

Software-Anforderung:
- Turbo-Pascal 3.0

ISB N 3-89090-455-6

Markt&Technik

4 001057 904552

DM 59,-
sFr 54,30
öS 460,20

90 455

Kassera

TURBO-PASCAL 3.0 auf dem CPC6128

Markt&Technik

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/

[FRA] Ce document a été préservé numériquement à des fins éducatives et d'études, et non commerciales.
[ENG] This document has been digitally preserved for educational and study purposes, not for commercial purposes.
[ESP] Este documento se ha conservado digitalmente con fines educativos y de estudio, no con fines comerciales.